Distributed Snapshots: Determining Global States of Distributed Systems

K. MANI CHANDY
University of Texas at Austin
and
LESLIE LAMPORT
Stanford Research Institute

This paper presents an algorithm by which a process in a distributed system determines a global state of the system during a computation. Many problems in distributed systems can be cast in terms of the problem of detecting global states. For instance, the global state detection algorithm helps to solve an important class of problems: stable property detection. A stable property is one that persists: once a stable property becomes true it remains true thereafter. Examples of stable properties are "computation has terminated," "the system is deadlocked" and "all tokens in a token ring have disappeared." The stable property detection problem is that of devising algorithms to detect a given stable property. Global state detection can also be used for checkpointing.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; *distributed databases*; *network operating systems*; D.4.1 [**Operating Systems**]: Process Management—*concurrency*; *deadlocks*; *multiprocessing/multiprogramming*; *mutual exclusion*; *scheduling*; *synchronization*; D.4.5 [**Operating Systems**]: Reliability—*backup procedures*; *checkpoint/restart*; *fault-tolerance*; *verification*

General Terms: Algorithms

Additional Key Words and Phrases: Global States, Distributed deadlock detection, distributed systems, message communication systems

1. INTRODUCTION

This paper presents algorithms by which a process in a distributed system can determine a global state of the system during a computation. Processes in a distributed system communicate by sending and receiving messages. A process can record its own state and the messages it sends and receives; *it can record nothing else.* To determine a global system state, a process p must enlist the

This work was supported in part by the Air Force Office of Scientific Research under Grant AFOSR 81-0205 and in part by the National Science Foundation under Grant MCS 81-04459.
Authors' addresses: K. M. Chandy, Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712; L. Lamport, Stanford Research Institute, Menlo Park, CA 94025.

cooperation of other processes that must record their own local states and send the recorded local states to p. All processes cannot record their local states at precisely the same instant unless they have access to a common clock. We assume that processes do not share clocks or memory. The problem is to devise algorithms by which processes record their own states and the states of communication channels so that the set of process and channel states recorded form a global system state. The global-state-detection algorithm is to be superimposed on the underlying computation: it must run concurrently with, but not alter, this underlying computation.

The state-detection algorithm plays the role of a group of photographers observing a panoramic, dynamic scene, such as a sky filled with migrating birds—a scene so vast that it cannot be captured by a single photograph. The photographers must take several snapshots and piece the snapshots together to form a picture of the overall scene. The snapshots cannot all be taken at precisely the same instant because of synchronization problems. Furthermore, the photographers should not disturb the process that is being photographed; for instance, they cannot get all the birds in the heavens to remain motionless while the photographs are taken. Yet, the composite picture should be meaningful. The problem before us is to define "meaningful" and then to determine how the photographs should be taken.

We now describe an important class of problems that can be solved with the global-state-detection algorithm. Let y be a predicate function defined on the global states of a distributed system D; that is, $y(S)$ is true or false for a global state S of D. The predicate y is said to be a *stable property* of D if $y(S)$ implies $y(S')$ for all global states S' of D reachable from global state S of D. In other words, if y is a stable property and y is true at a point in a computation of D, then y is true at all later points in that computation. Examples of stable properties are "computation has terminated," "the system is deadlocked," and "all tokens in a token ring have disappeared."

Several distributed-system problems can be formulated as the general problem of devising an algorithm by which a process in a distributed system can determine whether a stable property y of the system holds. Deadlock detection [2, 5, 8, 9, 11] and termination detection [1, 4, 10] are special cases of the stable-property detection problem. Details of the algorithm are presented later. The basic idea of the algorithm is that a global state S of the system is determined and $y(S)$ is computed to see if the stable property y holds.

Several algorithms for solving deadlock and termination problems by determining the global states of distributed systems have been published. Gligor and Shattuck [5] state that many of the published algorithms are incorrect and impractical. A reason for the incorrect or impractical algorithms may be that the relationships among local process states, global system states, and points in a distributed computation are not well understood. One of the contributions of this paper is to define these relationships.

Many distributed algorithms are structured as a sequence of phases, where each phase consists of a transient part in which useful work is done, followed by a stable part in which the system cycles endlessly and uselessly. The presence of stable behavior indicates the end of a phase. A phase is similar to a series of

iterations in a sequential program, which are repeated until successive iterations produce no change, that is, stability is attained. Stability must be detected so that one phase can be terminated and the next phase initiated [10]. The termination of a computational phase is not identical to the termination of a computation. When a computation terminates, all activities cease—messages are not sent and process states do not change. There may be activity during the stable behavior that indicates the end of a computational phase—messages may be sent and received, and processes may change state, but this activity serves no purpose other than to signal the end of a phase. In this paper, we are concerned with the detection of stable system properties; the cessation of activity is only one example of a stable property.

Strictly speaking, properties such as "the system is deadlocked" are not stable if the deadlock is "broken" and computation is reinitiated. However, to keep exposition simple, we shall partition the overall problem into the problems of (1) detecting the termination of one phase (and informing all processes that a phase has ended) and (2) initiating a new phase. The following is a stable property: "the kth computational phase has terminated," $k = 1, 2, \ldots$. Hence, the methods presented in this paper are applicable to detecting the termination of the kth phase for a given k.

In this paper we restrict attention to the problem of detecting stable properties. The problem of initiating the next phase of computation is not considered here because the solution to that problem varies significantly depending on the application, being different for database deadlock detection than for detecting the termination of a diffusing computation.

We have to present our algorithms in terms of a model of a system. The model chosen is not important in itself; we could have couched our discussion in terms of other models. We shall describe our model informally and only to the level of detail necessary to make the algorithms clear.

2. MODEL OF A DISTRIBUTED SYSTEM

A distributed system consists of a finite set of processes and a finite set of channels. It is described by a labeled, directed graph in which the vertices represent processes and the edges represent channels. Figure 1 is an example.

Channels are assumed to have infinite buffers, to be error-free, and to deliver messages in the order sent. (The infinite buffer assumption is made for ease of exposition: bounded buffers may be assumed provided there exists a proof that no process attempts to add a message to a full buffer.) The delay experienced by a message in a channel is arbitrary but finite. The sequence of messages received along a channel is an initial subsequence of the sequence of messages sent along the channel. The *state* of a channel is the sequence of messages sent along the channel, excluding the messages received along the channel.

A process is defined by a set of states, an initial state (from this set), and a set of events. An event e in a process p is an atomic action that may change the state of p itself and the state of *at most one* channel c incident on p: the state of c may be changed by the sending of a message along c (if c is directed away from p) or the receipt of a message along c (if c is directed towards p). An event e is defined by (1) the process p in which the event occurs, (2) the state s of p immediately

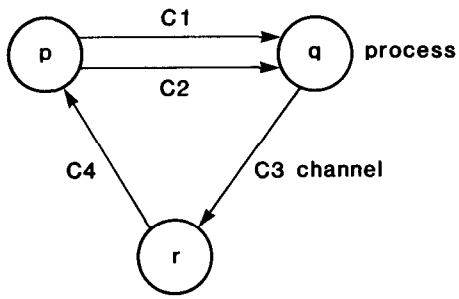

Fig. 1. A distributed system with processes p, q, and r and channels c1, c2, c3, and c4.

before the event, (3) the state s' of p immediately after the event, (4) the channel c (if any) whose state is altered by the event, and (5) the message M, if any, sent along c (if c is a channel directed away from p) or received along c (if c is directed towards p). We define e by the 5-tuple $\langle p, s, s', M, c\rangle$, where M and c are a special symbol, *null*, if the occurrence of e does not change the state of any channel.

A global state of a distributed system is a set of component process and channel states: the *initial* global state is one in which the state of each process is its initial state and the state of each channel is the empty sequence. The occurrence of an event may change the global state. Let $e = \langle p, s, s', M, c\rangle$ we say *e can occur in* global state S if and only if (1) the state of process p in global state S is s and (2) if c is a channel directed towards p, then the state of c in global state S is a sequence of messages with M at its head. We define a function *next*, where *next*(S, e) is the global state immediately after the occurrence of event e in global state S. The value of *next*(S, e) is defined only if event e can occur in global state S, in which case *next*(S, e) is the global state identical to S except that: (1) the state of p in *next*(S, e) is s'; (2) if e is a channel directed towards p, then the state of c in *next*(S, e) is c's state in S with message M deleted from its head; and (3) if c is a channel directed away from p, then the state of c in *next*(S, e) is the same as c's state in S with message M added to the tail.

Let $seq = (e_i: 0 \leq i \leq n)$ be a sequence of events in component processes of a distributed system. We say that seq is a *computation of the system* if and only if event e_i can occur in global state S_i, $0 \leq i \leq n$, where S_0 is the initial global state and

$$S_{i+1} = next(S_i, e_i) \qquad \text{for} \quad 0 \leq i \leq n.$$

An alternate model, based on Lamport [6], which views computations as partially ordered sets of events, is given in [7].

Example 2.1. To illustrate the definition of a distributed system, consider a simple system consisting of two processes p and q, and two channels c and c' as shown in Figure 2.

The system contains one *token* that is passed from one process to another, and hence we call this system the "single-token conservation" system. Each process has two states, s_0 and s_1, where s_0 is the state in which the process does not possess the token and s_1 is the state in which it does. The initial state of p is s_1 and of q is s_0. Each process has two events: (1) a transition from s_1 to s_0 with the

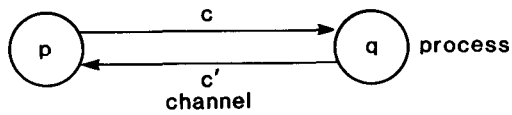

Fig. 2. The simple distributed system of Examples 2.1 and 2.2.

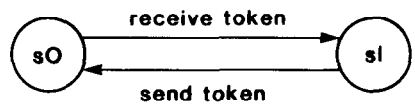

Fig. 3. State-transition diagram of a process in Example 2.1.

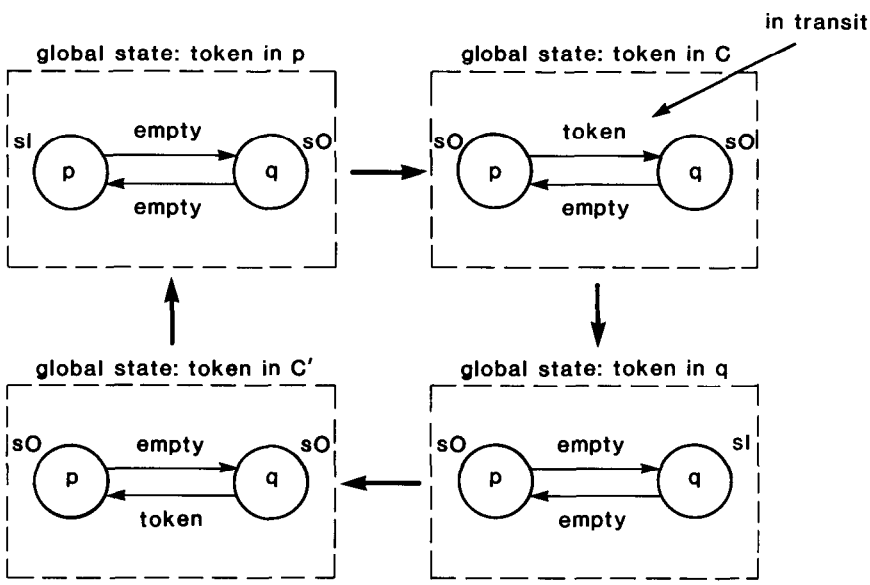

Fig. 4. Global states and transitions of the single-token conservation system.

sending of the token, and (2) a transition from s_0 to s_1 with the receipt of the token. The state-transition diagram for a process is shown in Figure 3. The global states and transitions are shown in Figure 4.

A system computation corresponds to a path in the global-state-transition diagram (Figure 4) starting at the initial global state. Examples of system computations are: (1) the empty sequence and (2) ⟨p sends token, q receives token, q sends token⟩. The following sequence is not a computation of the system: ⟨p sends token, q sends token⟩, because the event "q sends token" cannot occur while q is in the state s_0.

For brevity, the four global states, in order of transition (see Figure 4), will be called (1) in-p, (2) in-c, (3) in-q, and (4) in-c', to denote the location of the token. This example will be used later to motivate the algorithm. □

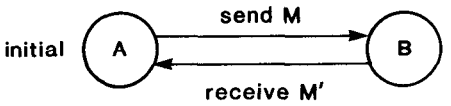

Fig. 5. State-transition diagram for process p in Example 2.2.

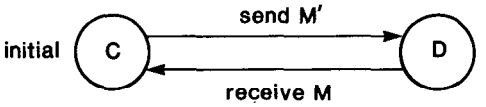

Fig. 6. State-transition diagram for process q in Example 2.2.

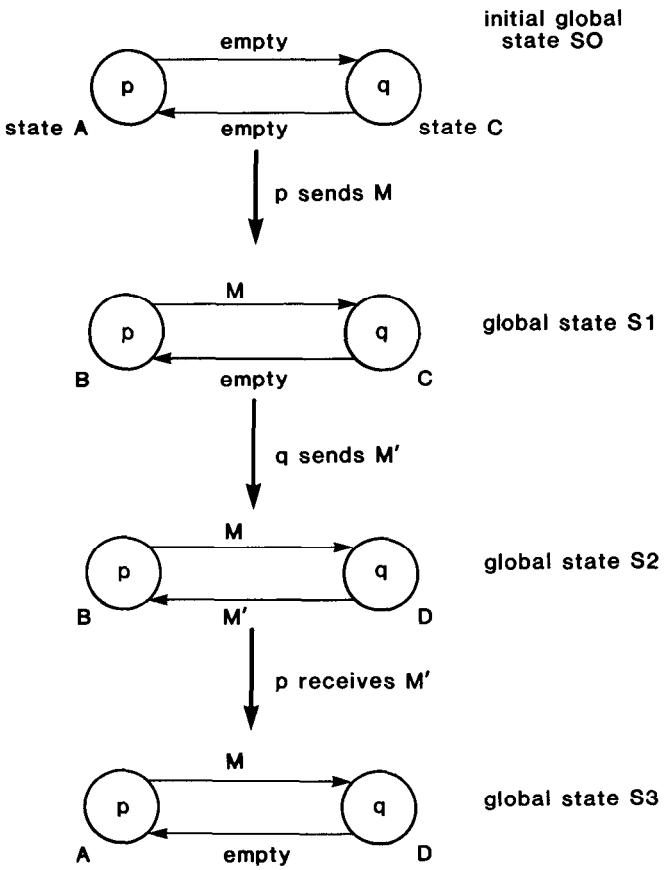

Fig. 7. A computation for Example 2.2.

Example 2.2. This example illustrates nondeterministic computations. Nondeterminism plays an interesting role in the snapshot algorithm.

In Example 2.1 there is exactly one event possible in each global state. Consider a system with the same topology as Example 2.1 (see Figure 2) but where the processes p and q are defined by the state-transition diagrams of Figures 5 and 6.

An example of a computation is shown in Figure 7. The reader should observe that there may be more than one transition allowable from a global state. For instance, events "p sends M" and "q sends M'" may occur in the initial global state, and the next states after these events are different. □

3. THE ALGORITHM

3.1. Motivation for the Steps of the Algorithm

The global-state recording algorithm works as follows: Each process records its own state, and the two processes that a channel is incident on cooperate in recording the channel state. We cannot ensure that the states of all processes and channels will be recorded at the same instant because there is no global clock; however, we require that the recorded process and channel states form a "meaningful" global system state.

The global-state recording algorithm is to be superimposed on the underlying computation, that is, it must run concurrently with, but not alter, the underlying computation. The algorithm may send messages and require processes to carry out computations; however, the messages and computation required to record the global state must not interfere with the underlying computation.

We now consider an example to motivate the steps of the algorithm. In the example we shall assume that we can record the state of a channel instantaneously; we postpone discussion of how the channel state is recorded. Let c be a channel from p to q. The purpose of the example is to gain an intuitive understanding of the relationship between the instant at which the state of channel c is to be recorded and the instants at which the states of processes p and q are to be recorded.

Example 3.1. Consider the single-token conservation system. Assume that the state of process p is recorded in global state in-p. Then the state recorded for p shows the token in p. Now assume that the global state transits to in-c (because p sends the token). Suppose the states of channels c and c' and of process q were recorded in global state in-c, so the state recorded for channel c shows it with the token and the states recorded for channel c' and process q show them not in possession of the token. The composite global state recorded in this fashion would show two tokens in the system, one in p and the other in c. But a global state with two tokens is unreachable from the initial global state in a *single-token* conservation system! The inconsistency arises because the state of p is recorded *before* p sent a message along c and the state of c is recorded *after* p sent the message. Let n be the number of messages sent along c before p's state is recorded, and let n' be the number of messages sent along c before c's state is recorded. Our example suggests that the recorded global state may be inconsistent if $n < n'$.

Now consider an alternate scenario. Suppose the state of c is recorded in global state in-p, the system then transits to global state in-c, and the states of c', p, and q are recorded in global state in-c. The recorded global state shows *no* tokens in the system. This example suggests that the recorded global state may be inconsistent if the state of c is recorded *before* p sends a message along c and the state of p is recorded *after* p sends a message along c, that is, if $n > n'$. □

We learn from these examples that (in general) a consistent global state requires

$$n = n'. \tag{1}$$

Let m be the number of messages received along c before q's state is recorded. Let m' be the number of messages received along c before c's state is recorded. We leave it up to the reader to extend the example to show that consistency requires

$$m = m'. \tag{2}$$

In every state, the number of messages received along a channel cannot exceed the number of messages sent along that channel, that is,

$$n' \geq m'. \tag{3}$$

From the above equations,

$$n \geq m. \tag{4}$$

The state of channel c that is recorded must be the sequence of messages sent along the channel before the sender's state is recorded, excluding the sequence of messages received along the channel before the receiver's state is recorded—that is, if $n' = m'$, the recorded state of c must be the empty sequence, and if $n' > m'$, the recorded state of c must be the $(m' + 1)$st, . . . , n'th messages sent by p along c. This fact and eqs. (1)–(4) suggest a simple algorithm by which q can record the state of channel c. Process p sends a special message, called a *marker*, after the nth message it sends along c (and before sending further messages along c). The marker has no effect on the underlying computation. The state of c is the sequence of messages received by q after q records its own state and before q receives the marker along c. To ensure eq. (4), q must record its state, if it has not done so already, after receiving a marker along c and before q receives further messages along c.

Our example suggests the following outline for a global state detection algorithm.

3.2 Global-State-Detection Algorithm Outline

Marker-Sending Rule for a Process p. For each channel c, incident on, and directed away from p:

p sends one marker along c after p records its state and before p sends further messages along c.

Marker-Receiving Rule for a Process q. On receiving a marker along a channel c:

if q has not recorded its state **then**
  begin q records its state;
    q records the state c as the empty sequence
  end
else q records the state of c as the sequence of messages received along c after q's state was recorded and before q received the marker along c.

3.3 Termination of the Algorithm

The marker receiving and sending rules guarantee that if a marker is received along every channel, then each process will record its state and the states of all

incoming channels. To ensure that the global-state recording algorithm terminates in finite time, each process must ensure that (L1) no marker remains forever in an incident input channel and (L2) it records its state within finite time of initiation of the algorithm.

The algorithm can be initiated by one or more processes, each of which records its state spontaneously, without receiving markers from other processes; we postpone discussion of what may cause a process to record its state spontaneously. If process p records its state and there is a channel from p to a process q, then q will record its state in finite time because p will send a marker along the channel and q will receive the marker in finite time (L1). Hence if p records its state and there is a path (in the graph representing the system) from p to a process q, then q will record its state in finite time because, by induction, every process along the path will record its state in finite time. Termination in finite time is ensured if for every process q: q spontaneously records its state or there is a path from a process p, which spontaneously records its state, to q.

In particular, if the graph is strongly connected and at least one process spontaneously records its state, then all processes will record their states in finite time (provided L1 is ensured).

The algorithm described so far allows each process to record its state and the states of incoming channels. The recorded process and channel states must be collected and assembled to form the recorded global state. We shall not describe algorithms for collecting the recorded information because such algorithms have been described elsewhere [4, 10]. A simple algorithm for collecting information in a system whose topology is strongly connected is for each process to send the information it records along all outgoing channels, and for each process receiving information for the first time to copy it and propagate it along all of its outgoing channels. All the recorded information will then get to all the processes in finite time, allowing all processes to determine the recorded global state.

4. PROPERTIES OF THE RECORDED GLOBAL STATE

To gain an intuitive understanding of the properties of the global state recorded by the algorithm, we shall study Example 2.2. Assume that the state of p is recorded in global state S_0 (Figure 7), so the state recorded for p is A. After recording its state, p sends a marker along channel c. Now assume that the system goes to global state S_1, then S_2, and then S_3 while the marker is still in transit, and the marker is received by q when the system is in global state S_3. On receiving the marker, q records its state, which is D, and records the state of c to be the empty sequence. After recording its state, q sends a marker along channel c'. On receiving the marker, p records the state of c' as the sequence consisting of the single message M'. The recorded global state S^* is shown in Figure 8. The recording algorithm was initiated in global state S_0 and terminated in global state S_3.

Observe that the global state S^* recorded by the algorithm is not identical to any of the global states S_0, S_1, S_2, S_3 that occurred in the computation. Of what use is the algorithm if the recorded global state never occurred? We shall now answer this question.

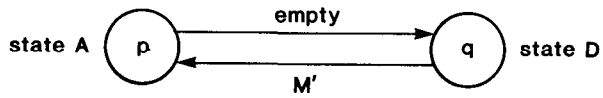

Fig. 8. A recorded global state for Example 2.2.

Let $seq = (e_i, 0 \leq i)$ be a distributed computation, and let S_i be the global state of the system immediately before event e_i, $0 \leq i$, in *seq*. Let the algorithm be initiated in global state S_ι and let it terminate in global state S_ϕ, $0 \leq \iota \leq \phi$; in other words, the algorithm is initiated after $e_{\iota-1}$ if $\iota > 0$, and before e_ι, and it terminates after $e_{\phi-1}$ if $\phi > 0$, and before e_ϕ. We observed in Example 2.2 that the recorded global state S^* may be different from all global states S_k, $\iota \leq k \leq \phi$.

We shall show that:

(1) S^* is reachable from S_ι, and
(2) S_ϕ is reachable from S^*.

Specifically, we shall show that there exists a computation *seq'* where

(1) *seq'* is a permutation of *seq*, such that S_ι, S^* and S_ϕ occur as global states in *seq'*,
(2) $S_\iota = S^*$ or S_ι occurs earlier than S^*, and
(3) $S_\phi = S^*$ or S^* occurs earlier than S_ϕ in *seq'*.

THEOREM 1. *There exists a computation* $seq' = (e'_i, 0 \leq i)$ *where*

(1) *For all i, where* $i < \iota$ *or* $i \geq \phi$: $e'_i = e_i$, and
(2) *the subsequence* $(e'_i, \iota \leq i < \phi)$ *is a permutation of the subsequence* $(e_i, \iota \leq i < \phi)$, *and*
(3) *for all i where* $i \leq \iota$ *or* $i \geq \phi$: $S'_i = S_i$, *and*
(4) *there exists some k,* $\iota \leq k \leq \phi$, *such that* $S^* = S'_k$.

PROOF. Event e_i in *seq* is called a *prerecording* event if and only if e_i is in a process p and p records its state *after* e_i in *seq*. Event e_i in *seq* is called a *postrecording* event if and only if it is not a prerecording event—that is, if e_i is in a process p and p records its state *before* e_i in *seq*. All events e_i, $i < \iota$, are prerecording events and all events e_i, $i \geq \phi$, are postrecording events in *seq*. There *may be* a postrecording event e_{j-1} before a prerecording event e_j for some j, $\iota < j < \phi$; this can occur only if e_{j-1} and e_j are in different processes (because if e_{j-1} and e_j are in the same process and e_{j-1} is a postrecording event, then so is e_j).

We shall derive a computation *seq'* by permuting *seq*, where all prerecording events occur before all postrecording events in *seq'*. We shall show that S^* is the global state in *seq'* after all prerecording events and before all postrecording events.

Assume that there is a postrecording event e_{j-1} before a prerecording event e_j in *seq*. We shall show that the sequence obtained by interchanging e_{j-1} and e_j must also be a computation. Events e_{j-1} and e_j must be on different processes. Let p be the process in which e_{j-1} occurs, and let q be the process in which e_j occurs. There cannot be a message sent at e_{j-1} which is received at e_j because (1)

if a message is sent along a channel c when event e_{j-1} occurs, then a marker must have been sent along c *before* e_{j-1}, since e_{j-1} is a postrecording event, and (2) if the message is received along channel c when e_j occurs, then the marker must have been received along c *before* e_j occurs (since channels are first-in-first-out), in which case (by the marker-receiving rule) e_j would be a postrecording event too.

The state of process q is not altered by the occurrence of event e_{j-1} because e_{j-1} is in a different process p. If e_j is an event in which q receives a message M along a channel c, then M must have been the message at the head of c before event e_{j-1}, since a message sent at e_{j-1} cannot be received at e_j. Hence event e_j can occur in global state S_{j-1}.

The state of process p is not altered by the occurrence of e_j. Hence e_{j-1} can occur after e_j. Hence the sequence of events $e_1, \ldots, e_{j-2}, e_j, e_{j-1}$ is a computation. From the arguments in the last paragraph it follows that the global state after computation $e_1, \ldots, e_j$ is the same as the global state after computation $e_1, \ldots, e_{j-2}, e_j, e_{j-1}$.

Let *seq** be a permutation of *seq* that is identical to *seq* except that e_j and e_{j-1} are interchanged. Then *seq** must also be a computation. Let $\bar{S}_i$ be the global state immediately before the ith event in *seq**. From the arguments of the previous paragraph,

$$\bar{S}_i = S_i \qquad \text{for all} \quad i \quad \text{where} \quad i \neq j.$$

By repeatedly swapping postrecording events that immediately follow prerecording events, we see that there exists a permutation *seq'* of *seq* in which

(1) all prerecording events precede all postrecording events,
(2) *seq'* is a computation,
(3) for all i where $i < \iota$ or $i \geq \phi$: $e'_i = e_i$, and
(4) for all i where $i \leq \iota$ or $i \geq \phi$: $S'_i = S_i$.

Now we shall show that the global state after all prerecording events and before all postrecording events in *seq'* is S^*. To do this, we need to show that

(1) the state of each process p in S^* is the same as its state after the process computation consisting of the sequence of prerecorded events on p, and
(2) the state of each channel c in S^* is (sequence of messages corresponding to prerecorded sends on c) − (sequence of messages corresponding to prerecorded receives on c).

The proof of the first part is trivial. Now we prove part (2). Let c be a channel from process p to process q. The state of channel c recorded in S^* is the sequence of messages received on c by q *after* q records its state *and before* q receives a marker on c. The sequence of messages sent by p along c before p sends a marker along c is the sequence corresponding to prerecorded sends on c. Part (2) now follows. □

Example 4.1. The purpose of this example is to show how the computation *seq'* is derived from the computation *seq*. Consider Example 2.2. The sequence

of events shown in the computation of Figure 7 is

e_0: p sends M and changes state to B (a postrecording event)

e_1: q sends M' and changes state to D (a prerecording event)

e_2: p receives M' and changes state to A (a postrecording event)

Since e_0, a postrecording event, immediately precedes e_1, a prerecording event, we interchange them, to get the permuted sequence seq':

e'_0: q sends M' and changes state to D (a prerecording event)

e'_1: p sends M and changes state to B (a postrecording event)

e'_2: p receives M' and changes state to A (a postrecording event)

In seq', all prerecording events precede all postrecording events. We leave it to the reader to show that the global state after e'_0 is the recorded global state.

5. STABILITY DETECTION

We now solve the stability-detection problem described in Section 1. We study the stability-detection problem because it is a paradigm for many practical problems, such as distributed deadlock detection.

A stability-detection algorithm is defined as follows:

Input: A stable property y

Output: A Boolean value *definite* with the property:
$(y(S_i) \rightarrow definite)$ and $(definite \rightarrow y(S_\phi))$

where S_i and S_ϕ are the global states of the system when the algorithm is initiated and when it terminates, respectively. (The symbol $\rightarrow$ denotes logical implication.)

The input to the algorithm is (the definition of) function y. During the execution of the algorithm the value $y(S)$ for some global state S may be determined by a process in the system by applying the *externally defined* function y to global state S. By the output of the algorithm being a Boolean value *definite* we mean that (1) some specially designated process (say p) enters and thereafter remains in some special state to symbolize an output of *definite* = *true*, and (2) p enters and remains in some other special state to symbolize an output of *definite* = *false*.

Definite = *true* implies that the stable property holds when the algorithm *terminates*. However, *definite* = *false* implies that the stable property does not hold when the algorithm is *initiated*. We emphasize that *definite* = *true* gives us information about the state of the system at the *termination* of the algorithm, whereas *definite* = *false* gives us information about the system state at the *initiation* of the algorithm. In particular, we cannot deduce from *definite* = *false* that the stable property does not hold at termination of the algorithm.

The solution to the stability detection problem is

```
begin
    record a global state S*;
    definite := y(S*)
end.
```

The correctness of the stability detection algorithm follows from the following facts:

(1) S^* is reachable from S_i,
(2) S_ϕ is reachable from S^* (Theorem 1), and
(3) $y(S) \rightarrow y(S')$ for all S' reachable from S (definition of a stable property).

ACKNOWLEDGMENTS

J. Misra's contributions in defining the problem of global state detection are gratefully acknowledged. We are grateful to E. W. Dijkstra and C. S. Scholten for their comments—particularly regarding the proof of Theorem 1. The outline of the current version of the proof was suggested by them. Dijkstra's note [3] on the subject provides colorful insight into the problem of stability detection. Thanks are due to C. A. R. Hoare, F. Schneider, and G. Andrews who helped us with detailed comments. We are grateful to Anita Jones and anonymous referees for suggestions.

REFERENCES

1. CHANDY, K. M., AND MISRA, J. Distributed computation on graphs: Shortest path algorithms. *Commun. ACM 25,* 11 (Nov. 1982), 833–837.
2. CHANDY, K. M., MISRA, J., AND HAAS, L. Distributed deadlock detection. *ACM Trans. Comput. Syst.* 1, 2 (May 1983), 144–156.
3. DIJKSTRA, E. W. The distributed snapshot of K. M. Chandy and L. Lamport. Tech. Rep. EWD 864a, Univ. of Texas, Austin, Tex., 1984.
4. DIJKSTRA, E. W., AND SCHOLTEN, C. S. Termination detection for diffusing computations. *Inf. Proc. Lett. 11,* 1 (Aug. 1980), 1–4.
5. GLIGOR, V. D., AND SHATTUCK, S. H. Deadlock detection in distributed systems. *IEEE Trans. Softw. Eng. SE-6,* 5 (Sep. 1980), 435–440.
6. LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21,* 7 (Jul. 1978), 558–565.
7. LAMPORT, L., AND CHANDY, K. M. On partially-ordered event models of distributed computations. Submitted for publication.
8. MAHOUD, S. A., AND RIORDAN, J. S. Software controlled access to distributed databases. *INFOR 15,* 1 (Feb. 1977), 22–36.
9. MENASCE, D., AND MUNTZ, R. Locking and deadlock detection in distributed data bases. *IEEE Trans. Softw. Eng. SE-5,* 3 (May 1979), 195–202.
10. MISRA, J., AND CHANDY, K. M. Termination detection of diffusing computations in communicating sequential processes. *ACM Trans. Program. Lang. Syst.* 4, 1 (Jan. 1982), 37–43.
11. OBERMARCK, R. Distributed deadlock detection algorithm. *ACM Trans. Database Syst. 7,* 2 (Jun. 1982), 187–208.

Received January 1984; revised September 1984; accepted 7 December 1984