Testing and Debugging Distributed Programs Using Global Predicates

S. Venkatesan and Brahma Dathan

Abstract—**Testing and debugging programs are more involved in distributed systems than in uniprocessor systems because of the presence of the communication medium and the inherent concurrency. Past research has established that predicate testing is an approach that can alleviate some of the problems in this area. However, checking whether a general predicate is true in a particular distributed execution appears to be a computationally hard problem. This paper considers a class of predicates called *conjunctive form predicates* (*CFP*) that is quite useful in distributed program development, but can be tested efficiently. We develop fully-distributed algorithms to test CFP's, prove that these algorithms are correct, and analyze them for their message complexity. The analysis shows that our techniques incur a fairly low overhead on the distributed system.**

Index Terms—**Distributed algorithms, distributed testing, distributed debugging, global predicates, asynchronous distributed systems, message complexity.**

I. Introduction

DISTRIBUTED SYSTEMS consist of several computer systems connected to each other by a communication network. Although they have several advantages over centralized systems, distributed systems are harder to design and program [9] because of the following reasons: First, a varying number of processes execute in parallel. Second, a process may update its variables independently or in response to the actions of the other processes. Third, problems specific to the development of distributed applications are not suitably reflected by known programming languages and software engineering environments [21].

Since distributed programs are inherently hard to design and develop, they are also difficult to test and debug. Testing and debugging are further complicated because communication delays are random. As a result, messages destined to a specific process may not maintain their relative order in repeated executions. This is true even if the external input to the system is the same. Also, the absence of a single point of control and a common clock severely hampers the ability to detect and examine errors. Finally, debugging is difficult as setting breakpoints and halting are nontrivial in a multiprocessor environment.

The focus of this paper is on the testing and debugging phases of distributed software development. To gain an understanding of the important issues addressed in this paper and the approach we take, consider the problem of distributed mutual exclusion, a topic of theoretical and practical significance [25]. Let the distributed system consist of n processes and a shared resource, perhaps, a printer. The nature of the resource demands that it must be allocated to at most one process at any given time. The mutual exclusion problem is to control the access to the resource by the processes satisfying the above property.

Assume that we are testing the implementation of a standard mutual exclusion algorithm [14]. Let each process P_i maintain a variable ME_i that is set to *true* if and only if P_i obtains mutual exclusion. To check if processes P_1 and P_2 are in the critical section at the "same time," we may test whether the predicate $ME_1 \wedge ME_2$ evaluates to true at "some time." If so, the mutual exclusion requirement has been violated and the execution exposes an error. On the other hand, if the predicate is false at all times during this execution, then a violation of the mutual exclusion requirement involving P_1 and P_2 did not occur in this execution.

The above example illustrates the usefulness of evaluating global predicates in identifying the presence of errors (bugs). Global predicates are useful in distributed debugging also [8], [27]. For example, during debugging the user may want to halt the execution at the "first time instant" when a predicate, say $Y_1 = 5 \wedge Y_2 > 8$, becomes true and examine the values of the other variables.

This paper presents a practical approach to testing and debugging distributed programs based on global predicates. Testing begins with the construction of global predicates. The distributed program is then run, and each process locally records information relevant to these predicates. After the run, the gathered information is used to evaluate the predicates to expose errors. The debugging stage is replay based (information gathered during the first run is used by the debugger during the subsequent reruns) and may involve the construction and evaluation of additional predicates.

The rest of the paper is organized as follows: Section II presents the distributed system model. We discuss global predicates in Section III. The major contribution of the paper, distributed algorithms to evaluate global predicates, appears in Section IV Section V compares our results with the related work. Finally Section VI concludes the paper.

Manuscript received December 1992; revised March 1994. Recommended by R. Taylor. This work was supported in part by the NSF under Grant CCR-9110177 and by the Texas Advanced Technology Program under Grant 9741-036. This paper was presented in part at the Thirtieth Annual Allerton Conference on Communication, Control, and Computing, October 1992.

S. Venkatesan is with the Computer Science Program, University of Texas at Dallas, Richardson, TX 75083 USA.

B. Dathan is with the Department of Computer Science, St. Cloud State University, St. Cloud, MN 56301 USA.

IEEE Log Number 9407724.

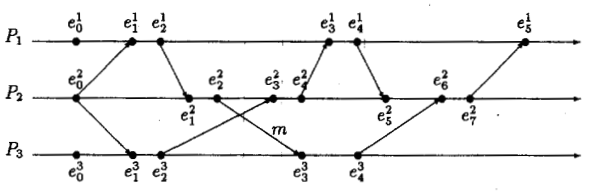


Fig. 1. Space-time diagram of an execution.

II. System Model

The distributed system is modeled as a set of n processes labeled $P_1, \cdots, P_n$. Processes communicate with each other by passing messages only. Each communication channel, which connects two processes, preserves FIFO message ordering. The system is *asynchronous*: there is no upper bound on the time taken by the processes to complete an instruction; nor are the message delivery times bounded. Eventually, however, processes will execute all instructions submitted for execution, and messages sent will be delivered to their destinations.

The salient features of an execution of a distributed program can be represented by a space-time diagram consisting of one horizontal line per process depicting its behavior. (Throughout the paper, we consider finite executions and, therefore, finite space-time diagrams.) The horizontal direction denotes time, which increases as we go from left to right. Message exchanges are shown by directed lines with the direction representing the direction of message transmission.

The Notion of Events: A process in execution is a sequence of events (the sequence maintaining the relative order of occurrence) taking place within it. An event is one of three types: receive event (receiving a message), local event (updating variables, accessing secondary memory, etc.) and send event (sending one message or broadcasting a message). Let e_j^i denote an event numbered j in P_i. We number events within each process with consecutive integers starting with 0.

Formal Definitions: A space-time diagram can formally be represented by a directed acyclic graph with exactly one node for each event. (Label each node with the id of the event it represents.) There is a directed edge from event e to event e' if (i) e and e' occur within the same process and e' occurs immediately after e or (ii) the receipt of a message sent during e starts event e'. This directed acyclic graph induces a partial order on the events [14].

We say that e is a *predecessor* of e' and e' is a *successor* of e, denoted by $e \prec e'$, if there exists a path from e to e'. Note that an event is a predecessor of itself and a successor of itself. If $e \prec e'$ and $e \neq e'$, we say that $e(e')$ is a proper predecessor (successor) of $e'(e)$. The fact that e is not a predecessor of e' is denoted by writing $e \not\prec e'$. In Fig. 1 (where the events are shown as bullets •), $e_0^1 \prec e_1^2$ and $e_1^1 \not\prec e_1^3$.

A *spectrum* is a sequence of consecutive events in a process. At its extreme, the spectrum may be just a single event or it may include all of the events in the process.

Let S_i be a spectrum in P_i. An event e_j^i in S_i is the last event in S_i if no other event in S_i is a successor of e_j^i; e_j^i is said to be first event in S_i if no other event in S_i is a predecessor of e_j^i. The last (first) event of S_i is denoted by $last(S_i)$ $(first(S_i))$. Since a spectrum may include all the events of a process, last and first are well defined.

A global state is a set of processor states, one state per process. A global state GS is *consistent* if GS "includes" the sending of message m' whenever the receipt of m' is "included" in GS. This notion of consistency is from Chandy and Lamport [6]. Let s_j^i be the process state of P_i immediately after executing event e_j^i. In Fig. 1, the global state (s_1^1, s_0^2, s_1^3) is consistent while the global state (s_3^1, s_3^2, s_2^3) is not consistent since s_3^1 includes the receipt of m while s_3^2 does not include the sending of m_3. A global state GS, defined as above, can also be represented by a *cut* C which is a set of events, one event per process, such that event $e \in$ C if and only if the processor state immediately after event e is a part of GS. A cut is *consistent* if and only if the corresponding global state is consistent. Thus, a consistent global state corresponds to a consistent cut and vice versa. In Fig. 1, the cut (e_1^1, e_0^2, e_1^3), which corresponds to global state (s_1^1, s_0^2, s_1^3), is consistent.

Let e_j^i be an event of process P_i. The *first consistent cut event* of e_j^i at process P_k, denoted by $\mathcal{FCCE}_k(e_j^i)$, is the earliest event e_l^k such that e_l^k and e_j^i can be in a consistent cut. If no event of P_k precedes $e_j^i (e_l^k \not\prec e_j^i$ for all $l \geq 0)$, then $\mathcal{FCCE}_k(e_j^i) = e_0^k$; if some event of P_k precedes e_j^i, then $\mathcal{FCCE}_k(e_j^i) \prec e_j^i$ and for any e_m^k such that $\mathcal{FCCE}_k(e_j^i) \prec e_m^k$, $e_m^k \not\prec e_j^i$. For example, in Fig. 1, $\mathcal{FCCE}_2(e_3^3) = e_2^2$, $\mathcal{FCCE}_1(e_3^3) = e_2^1$, $\mathcal{FCCE}_1(e_2^3) = e_0^1$, and $\mathcal{FCCE}_2(e_4^3) = e_2^2$.

Observation 1: If e and e' are events on some process P_i such that $e \prec e'$, then $\mathcal{FCCE}_k(e) \prec \mathcal{FCCE}_k(e')$ for all processes P_k. ■

The *last consistent cut event* of e_j^i at process P_k, denoted by $\mathcal{LCCE}_k(e_j^i)$, is the latest event of P_k such that e_j^i and $\mathcal{LCCE}_k(e_j^i)$ belong to a consistent cut. $\mathcal{LCCE}_k(e_j^i)$ is the latest event e_l^k such that if e_{j+1}^i (the next event of e_j^i) exists, then $e_{j+1}^i \not\prec e_l^k$; if e_j^i is the last event of P_i, then $\mathcal{LCCE}_k(e_j^i)$ is the last event of P_k. In Fig. 1, $\mathcal{LCCE}_2(e_3^3) = e_5^2$, $\mathcal{LCCE}_1(e_3^3) = e_4^1$, $\mathcal{LCCE}_1(e_4^3) = e_5^1$, and $\mathcal{LCCE}_3(e_4^1) = e_4^3$.

Observation 2: If e and e' are events on some process P_i such that $e \prec e'$, then $\mathcal{LCCE}_k(e) \prec \mathcal{LCCE}_k(e')$ for all processes P_k.

The *consistency set* of an event e_j^i at process P_k is the spectrum S_k with $first(S_k) = \mathcal{FCCE}_k(e_j^i)$ and $last(S_k) = \mathcal{LCCE}_k(e_j^i)$. In Fig. 1, the consistency set of e_3^3 at $P_1 = \{e_2^1, e_3^1, e_4^1\}$.

The notion of predecessor, successor, etc. can be extended to consistent cuts. Let Σ_i and Σ_j be consistent cuts. $\Sigma_i(\Sigma_j)$ is said to be a predecessor (successor) of $\Sigma_j(\Sigma_i)$, expressed by writing $\Sigma_i \prec \Sigma_j$, if no component of Σ_j is a proper predecessor of the corresponding component of Σ_i. If $\Sigma_i \prec \Sigma_j$ and $\Sigma_i \neq \Sigma_j$, we say that $\Sigma_i(\Sigma_j)$ is a proper predecessor (successor) of $\Sigma_j(\Sigma_i)$. The fact that Σ_i is not a predecessor of Σ_j is denoted by writing $\Sigma_i \not\prec \Sigma_j$. A consistent cut that is a predecessor (successor) of all of the other consistent cuts is called the *initial (final)* consistent cut. In Fig. 1, (e_0^1, e_0^2, e_0^3) is the initial consistent cut and (e_5^1, e_7^2, e_4^3) is the final consistent cut.

Consistent Cut Lattice: Given a space-time diagram, it may not always be possible to say how the global execution

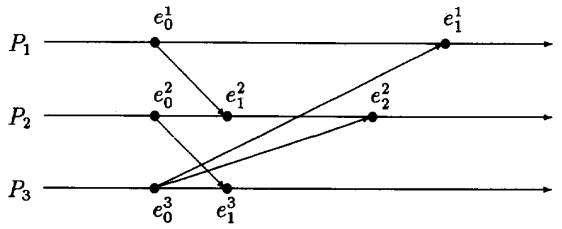


Fig. 2. A sample execution.

actually occurred. For example, consider the execution shown in Fig. 2. It is not clear if event e_1^3 or e_1^1 occurred first. Similar comments apply to many other pairs of events such as e_2^2 and e_1^3. Imposing a total order on the events by ordering such events introduces temporal relationships not present during the execution.

A single execution of a distributed algorithm can be captured by a single space-time diagram, but a space-time diagram represents several possible global executions. Thus many different global executions may result in the same space-time diagram. A clearer picture of the set of all such possible executions can be had by drawing a *global consistent cut lattice* or simply a *lattice*.

A lattice is a directed acyclic graph with exactly one node for each consistent cut. The lattice for the space-time diagram in Fig. 2 is shown in Fig. 3. Each node, shown by a circle, represents one consistent cut of the system. Each consistent cut is labeled by a sequence of integers, one for each process, such that if e_j^i is a component of the consistent cut, then the i^{th} component of the label is j. For example, the node labeled (120) represents the consistent cut (e_1^1, e_2^2, e_0^3). We draw an arc from Σ_i to Σ_j if and only if the system can reach Σ_j from Σ_i when some process performs its next event. In our example, from consistent cut (e_0^1, e_0^2, e_0^3), the system may change to (e_1^1, e_0^2, e_0^3) if e_1^1 occurs next, and this is represented by an arc from node (000) to (100). Similarly, arcs are drawn from (000) to (010) and (001) since the system may change from (e_0^0, e_0^2, e_0^3) to (e_0^1, e_1^2, e_0^3) or (e_0^1, e_0^2, e_1^3). The node that corresponds to the initial consistent cut does not have any arcs directed to it, and the node that represents the final consistent cut does not have any outgoing arcs.

For a given execution, the total number of consistent cuts is, in general, very large and may be exponential in the number of processes [8]. Approaches that examine each consistent cut of an execution are quite time-consuming and are impractical in distributed systems with a large number of processes. (The concept of lattice is used in this paper for ease of exposition only.) Our approach, described in Sections 3 and 4, examines the consistent cuts implicitly without an explicit examination of each possible consistent cut.

III. The Notion of Conjunctive Form Predicates

In this section, we consider the following issues related to testing and debugging distributed programs.

1) How do we ensure that a distributed program execution is indeed correct? While individual programs in the system may appear to work correctly, collectively the execution may be in error. For example, in a mutual exclusion program, two processes P_1 and P_2 may both obtain permission to enter their critical sections, which is an error, but observing the behavior of P_1 or P_2 in isolation does not reveal this bug.
2) Since the behavior of any program in the system is, in general, related to the actions of the other programs in the system, conventional debugging tools used in centralized, sequential systems are inadequate.

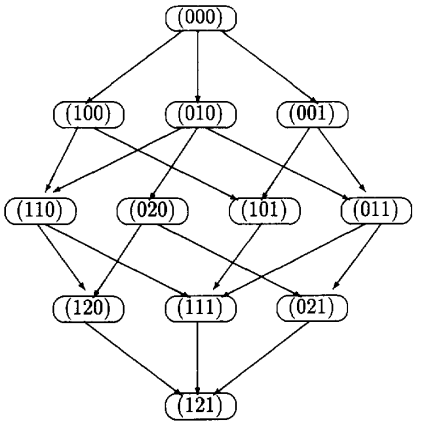


Fig. 3. Global consistent cut lattice of the execution of Fig. 2.

Past research [8], [27] has shown that a distributed program execution can be analyzed and questions related to its correctness answered by using the notion of *global predicates*. They can be used to ensure that certain "good events" occur and that certain "bad events" do not take place. Consider some examples. (In the following, subscripts of variables are used to denote the process to which they belong.)

Example: In a mutual exclusion program, we would like to ensure that two processes P_1 and P_2 do not execute within the critical section at the same time. This corresponds to the condition $not(ME_1 \wedge ME_2)$ for all consistent cuts in the space-time diagram.

Example: Consider a replicated database system that employs the quorum consensus protocol (see [3] for a description) for replica control. Let the system consist of 3 sites 1, 2, and 3 each storing a copy of an item x and each having a single vote. If all sites vote to permit write access on x to a transaction T_i executing on site 3, we may test that the predicate $(\text{VOTED\_FOR}_1(x) = T_i) \wedge (\text{VOTED\_FOR}_2(x) = T_i) \wedge (\text{VOTED\_FOR}_3(x) = T_i) \wedge (\text{VOTES\_RECEIVED}_3(T_i, x) = 3)$ holds in all possible executions of a particular test.

Example: A distributed database system may employ the 3-Phase Commit protocol (see description in [3]) to commit a transaction. If we expect a transaction T_i (executing on 3 sites) to commit in a particular test, we might check that the predicate $(\text{COMMITTED}_1(T_i) \vee \text{COMMITTABLE}_1(T_i)) \wedge (\text{COMMITTED}_2(T_i) \vee \text{COMMITTABLE}_2(T_i)) \wedge (\text{COMMITTED}_3(T_i) \vee \text{COMMITTABLE}_3(T_i))$ is true.

Predicates of the above form can be used for distributed program development by employing the following steps.

1) Perform a test using some test data $\mathcal{T}$.
2) Analyze the distributed execution in Step 1 using global predicates.
3) If Step 2 indicates error conditions, debug the programs. Just as in centralized systems, the debugging phase may involve repeating the test several times using the same test data $\mathcal{T}$ to gather more information about the bug and employing more predicates to pinpoint the actual cause of error.

The predicates used for testing and debugging may be evaluated as the test progresses. Such a strategy is termed *on-line evaluation*. A strategy that evaluates predicates after the test run is called *off-line evaluation*. There are advantages and disadvantages for both approaches, and they are discussed in §5. In this paper, we use the off-line evaluation approach.

It appears that given a general predicate, any predicate evaluation strategy, regardless of whether it is off-line or on-line, would, in general, need to check every consistent cut in the lattice. To justify this belief, consider a predicate Φ that represents an error condition. Assume that in a particular run, this predicate is never true for any consistent cut. But there seems no way to infer this without actually testing the condition for each consistent cut.

Since the number of consistent cuts in the lattice can be exponential in the number of processes, general predicate testing appears to be impractical for reasonably big systems.

In many situations, however, testing can be performed using predicates that are less general. These predicates are written as the conjunction of expressions $C_1, \cdots, C_n$, where each conjunct C_i is any boolean expression that is restricted to constants and/or variables local to a single process P_i. Such expressions can be used to test a variety of conditions. (See the examples given earlier in this section.)

Predicates having the above form are called *Conjunctive Form Predicates* (CFP). CFP's have been used in debugging [12], [20]. An advantage of CFP is that each process P_i can perform an independent evaluation, without communicating to any other process, of the sub-expression C_i at every event in P_i. This gives a set of spectra on P_i where C_i is true. (If Φ does not have a conjunct for one more processes, then we introduce a dummy conjunct that has the value *true* for all events of such processes.) We then analyze the spectra at all processes to check the global predicate.

The evaluation of a given CFP Φ depends on the purpose of Φ. (From now on, we use $\Phi = C_1 \wedge \ldots \wedge C_n$ to denote a CFP.) Obviously, if it represents an error condition, Φ must not be true at any time, so it may be sufficient to check if Φ is true in any consistent cut. For debugging purposes, however, one might also want to determine when Φ becomes true for the first time. On the other hand, if Φ stands for a good condition, we might want to ensure that Φ holds in all consistent cuts. In a typical development scenario, the user may have occasion to use an arbitrary predicate Φ in one or more of the following ways. (Note that the actual predicates to be used in a specific environment is dependent on the application, and, therefore, we do not discuss methods for constructing the predicates.)

POSSIBLY(Φ) is true if there exists a consistent cut where the predicate Φ holds. If Φ represents an error condition, then POSSIBLY(Φ) represents a possible occurrence of an error.

ALWAYS(Φ) is true if Φ holds in every consistent cut. In this case, Φ may be an invariant. It is easy to see that ALWAYS(Φ) $\equiv \neg$ POSSIBLY($\neg\Phi$).

DEFINITELY(Φ) is true if in every path from the initial consistent cut to the final consistent cut in the lattice, Φ holds at least in one consistent cut in the path. Intuitively, Φ represents something good that we desire to be true irrespective of the actual progress of the execution.

Consider the lattice in Fig. 3. Assume that some CFP Φ is true in consistent cuts (110), (020), (111), and (021). Any path from the initial consistent cut, (000), to the final consistent cut, (121), must pass through one of the consistent cuts in the set {(110), (020), (111), (021) }, making DEFINITELY(Φ) true for the execution.

FIRST(Φ): Assume that POSSIBLY(Φ) is true and let $\mathcal{S}$ be the set of consistent cuts in which Φ holds. Then, FIRST(Φ) is the unique consistent cut in $\mathcal{S}$ such that all other consistent cuts in $\mathcal{S}$ are its successors. If Φ is an error condition, then for debugging, the user may want to bring the computation to the first consistent cut where Φ holds.

Due to the special nature of CFP's, we are guaranteed that FIRST(Φ) is well defined. To see this, consider the example given above for DEFINITELY(Φ) where Φ holds in consistent cuts (110), (020), (111), and (021). Let $\Phi = C_1 \wedge C_2 \wedge C_3$. Since Φ holds in consistent cut (110), C_1 holds in event e_1^1, C_2 holds in event e_1^2, and C_3 holds in event e_0^3. Similarly, considering (020) C_1 holds in event e_0^1, C_2 holds in event e_2^2, and C_3 holds in event e_0^3. Surely, then Φ holds in consistent cut (010). Similarly, considering consistent cuts (111) and (021), we can see that Φ holds in consistent cut (011). The reader can also verify that Φ holds in consistent cuts (120) and (121). Thus the consistent cut (010) is a unique consistent cut that is a predecessor of all consistent cuts in which Φ holds.

The above argument can be generalized to show that FIRST(Φ) is always well defined for CFP's [17].

LAST(Φ): This is same as FIRST(Φ) except that we are interested in finding the consistent cut Σ such that all consistent cuts in which Φ holds are predecessors of Σ. If Φ denotes an error condition, then the user may want to find when and how the error "disappears."

As in the case of FIRST(Φ), we can show that LAST(Φ) is well defined. (Observation 2 is useful for this purpose.)

In the next section, we develop distributed algorithms to test all of the above predicates. As mentioned earlier, they are off-line algorithms that make use of the space-time diagram. Therefore, before discussing our techniques for predicate evaluation, we discuss how the space-time diagram is represented in a distributed system.

Representation of the Space-time Diagram: The space-time diagram is maintained in a distributed fashion with each process storing information about events taking place within it. When a message m is received, the process records it as a receive event and stores the id of the sender of m. When

a message is sent, the process marks a send event and stores the id of the destination. Local events need not be logged; it is sufficient if the space-time diagram provides just enough information to permit an off-line evaluation of the predicates. For each CFP Φ, we make an entry in the space-time diagram whenever the truth value of C_i changes.

It is desirable to store the space-time diagram in main memory to reduce the time overhead of storing the structure. If the test is long, however, it may be necessary to transfer the space-time diagram from main memory to secondary storage periodically since the main memory may be small, or the system may fail (a hardware failure or a faulty process may hang the system) leading to loss of information. Obviously, the loss of information can be reduced by increasing the frequency of such transfers from primary to secondary memory.

It is interesting to see the storage cost involved in maintaining the space-time diagram. Each event needs to be identified as a receive or send event requiring 1 bit. In addition the sender/destination process id is also stored. Assuming that processes can be represented using 8 bits (256 processes), each send/receive event can be represented using 9 bits. Additional storage is required for storing the identity of the predicate if several predicates are evaluated simultaneously.

Consider a one-hour test of a mutual exclusion algorithm involving 100 processes. Assume that a mutual exclusion request is generated every second for a total of 3600 requests. Each request may require the transmission of 200 messages, the test could involve the evaluation of 100 predicates, and assume that on the average, the predicate changes once per send/receive event, requiring 1 bit for each predicate for each send/receive event. Then, the total storage requirement per process would be less than 0.2 Mega bytes. Since the cost of primary/secondary storage has been falling sharply, this does not appear to be a particularly unreasonable requirement even in a personal computer, and the space-time diagram could be maintained in primary storage for speed up.

Spectra on a Process: Consider a CFP Φ and a sequence of events $< e_j^i, \ldots, e_l^i >$ in the space-time diagram such that C_i is true for all events $e_k^i, j \leq k \leq l$ and C_i is false for e_{j-1}^i and e_{l+1}^i, if they exist. All such maximal sequences of events form a set of spectra on P_i such that C_i is true for all events for all spectra in the set. It is convenient in our discussions to impose a total order on this set. We say that $S_{i_1}^i$ is a predecessor of $S_{i_2}^i$, written $S_{i_1}^i \prec S_{i_2}^i$, if and only if $last(S_{i_1}^i) \prec first(S_{i_2}^i)$. The spectrum that has no predecessor is called the *first* spectrum. The spectrum that has no successor is called the *last* spectrum.

We now turn our attention to the development of distributed algorithms for the evaluation of global predicates.

IV. Algorithms for Evaluating Global Predicates

Given a CFP Φ, algorithms to evaluate POSSIBLY(Φ), DEFINITELY(Φ), ALWAYS(Φ), FIRST(Φ), and LAST(Φ) are presented in this section. The algorithms are based on *projections of spectra.*

Let S_i be a spectrum at site P_i. The *projection* of S_i on process P_j, denoted by $\pi(S_i \rightarrow P_j)$ is a spectrum S_j at P_j, possibly empty, such that $first(S_j) = \mathcal{FCCE}_j(first(S_i))$ and $last(S_j) = \mathcal{LCCE}_j(last(S_i))$. Intuitively, $\pi(S_i \rightarrow P_j)$ is the spectrum S_j at P_j consisting of all those events of P_j that are in the consistency set of at least one event of S_i. Thus, $\pi(S_i \rightarrow P_j)$ consists of those events of P_j that can co-exist with an event of S_i. It is easy to see that $\pi(S_i \rightarrow P_j)$ results in a single spectrum (possibly empty) at P_j. From the definition of $\mathcal{FCCE}$ and $\mathcal{LCCE}$, we can see that $\pi(S_i \rightarrow P_i) = S_i$.

Let S_1^i and S_2^i be two spectra at P_i. The intersection of S_1^i and S_2^i is the spectrum S_3^i at P_i, possibly empty, such that e is included in S_3^i if and only if e is included in both S_1^i and S_2^i. The intersection operation is denoted by $\cap$. Informally, $S_1^i \cap S_2^i$ is a spectrum consisting of all those events that are common to S_1^i and S_2^i.

Let $S_1, \cdots, S_n$ be spectra on processes $P_1, \cdots, P_n$ respectively, and let $I_i = \bigcap_{j=1}^n \pi(S_j \rightarrow P_i)$. I_i consists of those events of S_i that are in the consistency set of at least one event of S_1, one event of S_2, $\cdots$, one event of S_n. Computing the intersection of projections is a fundamental part of our algorithms, and we first present algorithms for computing I_i for each process P_i.

A. Computing the Intersection of Projections

The Main Idea: Consider process P_i. I_i, the intersection of all of the projections at P_i, is initially set to S_i since I_i cannot be a super set of S_i. We then systematically remove those events of I_i that are not in the consistency set of any event of the spectrum of some process $P_j, j \neq i$. (Removing some events from I_i is equivalent to moving $first(I_i)$ in the "forward direction" or moving $last(I_i)$ in the "backward direction.") The process of removing events from I_i is performed in $n-1$ iterations. To separate the salient features of the algorithm from the details of the distributed implementation, we first present a sequential algorithm and prove its correctness. A distributed implementation is presented in Section IV-A2).

1) A Sequential Algorithm: The algorithm consists of two phases.

Phase 1: In the first phase (the initialization phase), we set I_i to $\pi(S_i \rightarrow P_i)$. (Recall that $\pi(S_i \rightarrow P_i)$ is S_i itself.) Let NSM_{ij}^l be the total number of messages sent by P_i to P_j up to and including event $last(I_i)$ of P_i, and let NMR_{ij}^l be the total number of messages received by P_i from P_j till event $last(I_i)$ of P_i. (The super script l denotes that event $last(I_i)$ is used.) Similarly, NSM_{ij}^f and NMR_{ij}^f are defined where the super script f denotes that event $first(I_i)$ is used. Note that the space-time diagram of P_i is sufficient for computing the values $\text{NSM}_{ij}^l, \text{NMR}_{ij}^l, \text{NSM}_{ij}^f$ and NMR_{ij}^f.

Phase 2: The second phase consists of $n-1$ iterations. During each iteration, for each pair of processes, we check for "inconsistencies" between the spectra $I_1, \cdots, I_n$. This process is performed in two steps.

Step 1: During each iteration, if the number of messages received by P_i from P_j till $last(I_i)$ is more than the number of messages sent by P_j to P_i till P_j's event $last(I_j)$ (that is, if $\text{NMR}_{ij}^l > \text{NSM}_{ji}^l$), it is clear that $last(I_i)$ "reflects" the receipt of some messages from P_j, but $last(I_j)$ does not "reflect" the sending of those messages. Thus, $last(I_i)$ cannot co-exist with any event of I_j. For example, see Fig. 4. In that

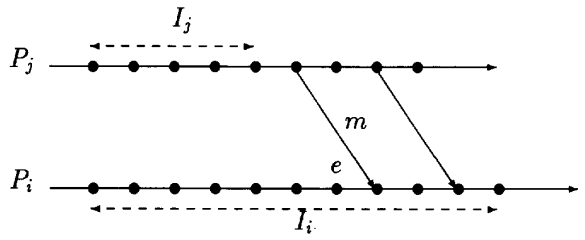


Fig. 4. Situation when $last(I_i)$ is updated in Step 1.

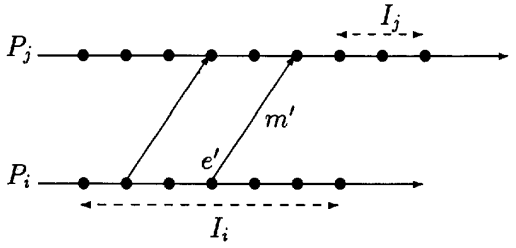


Fig. 5. Situation when $first(I_i)$ is updated in Step 2.

figure, $last(I_j)$ does not reflect sending of m, while $last(I_i)$ reflects the receipt of m. Clearly, $last(I_i)$ and $last(I_j)$ cannot be part of the same consistent cut. In this case, I_i must "shrink" and $last(I_i)$ must be updated so that the updated event $last(I_i)$ is in the consistency set of at least one event of I_j. Let e be the latest event of P_i such that the number of messages received by P_i from P_j till e is equal to NSM_{ji}^l. Clearly, such an event e exists, and $last(I_i)$ is set to e if e occurs before $last(I_i)$. Thus, $last(I_i)$ is updated in the "backward direction." (That is, the new value of $last(I_i)$ is not a proper predecessor of its old value.)

Step 2: If $\text{NMR}_{ji}^f > \text{NSM}_{ij}^f$, it is clear that $first(I_j)$ incorporates the receipt of a message (from P_i) that is not "reflected" by event $first(I_i)$. For example, in Fig. 5, the effect of receiving m' is incorporated into the event $first(I_j)$, while P_i has not sent m' during or till event $first(I_i)$. Thus, those events of I_i that do not "know" about sending of m' must be removed from I_i and $first(I_i)$ must be updated in the "forward direction." Let e' be the earliest event of P_i such that the number of messages sent by P_i to P_j till P_i's event e' is equal to NMR_{ji}^f. Now, $first(I_i)$ is changed to e'.

Steps 1 and 2 are performed for each ordered pair of processes, and $first(I_i)$ and $last(I_i)$ are updated for each process P_i. Intuitively, steps 1 and 2 "remove inconsistencies." This completes one iteration.

Several iterations are needed since there may be a situation where I_i must be updated in a cascaded manner to remove "inconsistencies." For example, in Fig. 6, m_1 is not sent by P_j during or before $last(I_j)$ and, for consistency, the effect of receiving m_1 must not be visible in any event of I_k for each process P_k. Since m_2 incorporates the effect of m_1, no event of I_k must incorporate message m_2 for each process P_k. Proceeding in this manner, it is easy to see that the effect of receiving message m_{n-1} of Fig. 6 must not be visible in

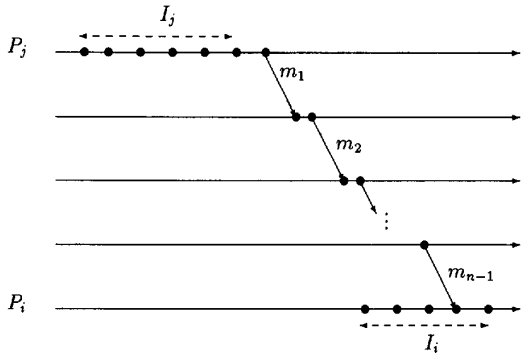


Fig. 6. Situation when $last(I_i)$ must be updated in a cascaded manner.

```
procedure pr_intersection;
{ this procedure finds the intersection of projections of spectra }
begin
Phase 1 (* initialization phase *)
    I_i ← S_i for all P_i;
    (* first(I_i) ← first(S_i); last(I_i) ← last(S_i) *)
    for each ordered pair of processes (P_i, P_j) do
        NMS^l_ij ← number of messages sent by P_i to P_j till event last(I_i);
        NMR^l_ij ← number of messages received by P_i from P_j till event last(I_i);
        NMS^f_ij ← number of messages sent by P_i to P_j till event first(I_i);
        NMR^f_ij ← number of messages received by P_i from P_j till event first(I_i);
    enddo;
Phase 2
    for ITERATIONS ← 1 to n − 1 do
        for each ordered pair of processes (P_i, P_j) do
        Step 1
            if NMR^l_ij > NMS^l_ji then (* last(I_i) cannot "co-exist" with last(I_j) *)
                e ← latest event of P_i such that NMS^l_ji = the number of messages
                    received by P_i from P_j till e;
                if e occurs before last(I_i) then last(I_i) = e;
                endif;
            endif;
        Step 2
            if NMR^f_ji > NMS^f_ij then (* first(I_i) cannot "co-exist" with first(I_j) *)
                e' ← earliest event of P_i such that NMR^f_ji = the number of messages
                    sent by P_i to P_j till e';
                if e' occurs after first(I_i) then first(I_i) = e';
                endif;
            endif;
        enddo; { inner for loop }
        (* if last(I_i) ≺ first(I_i) then I_i = φ and we stop *)
        recompute NMS^l_ij, NMR^l_ij, NMS^f_ij, and NMR^f_ij for all ordered pairs of processes (P_i, P_j);
    enddo;
end;
```

Fig. 7. Algorithm *pr_intersection*.

any event of I_i and $last(I_i)$ must be updated (in the backward direction) if each event of I_i is to be in the consistency set of at least one event of I_j.

After the first iteration, all inconsistencies between each pair of processes (that do not involve an intermediate process) are removed. After two iterations, all inconsistencies between each pair of processes that involve one intermediate process are removed. Thus, after $n-1$ iterations, all inconsistencies between any each pair of processes that involve any combination of intermediate processes are removed, and $n-1$ iterations are sufficient.

A formal description of the algorithm appears in Fig. 7, and its correctness is established below.

Theorem 1: At the end of the second phase of algorithm *pr_intersection*, if $I_i \neq \phi$ for all P_i, then $I_i = \cap_{j=1}^{n} \pi(S_j \rightarrow P_i)$.

Proof: Let $I_i' = \cap_{j=1}^{n} \pi(S_j \rightarrow P_i)$ for each P_i. Assume that $I_i' \neq I_i$ for some P_i. Thus, either $last(I_i') \neq last(I_i)$

or $first(I_i') \neq first(I_i)$. First assume the former. There are two cases to consider.

Case i): $last(I_i') \prec last(I_i)$ and $last(I_i) \neq last(I_i')$. Since $I_i' = \cap_{j=1}^{n} \pi(S_j \rightarrow P_i)$, there exists a processor $P_k, k \neq i$ such that $last(\pi(S_k \rightarrow P_i)) = last(I_i')$. It is easy to see from the definition of $\mathcal{LCCE}$ that there exists an event e of P_k and e' of P_i such that (1) e occurs after $last(S_k)$, (2) e' occurs immediately after $last(I_i')$, and (3) $e \prec e'$.

Since $e \prec e'$, there exists a sequence of (at most $n-2$) events $e_{a_1}, e_{a_2}, \cdots, e_{a_s} (s \leq n-2)$ on distinct processes $P_{b_1}, P_{b_2}, \cdots, P_{b_s}$, respectively, such that 1) $e \prec e_{a_1}$, 2) $e_{a_t} \prec e_{a_{t+1}}$ for all $1 \leq t < s$, and 3) $e_{a_s} \prec e'$. At the beginning of the first iteration of the algorithm, $\mathrm{NSM}_{kb_1}^{l}$ is one less than the number of messages received by P_{b_1} from P_k till P_{b_1}'s event e_{a_1}. Thus, at the end of the first iteration $last(I_{b_1})$ is set to a proper predecessor of e_{a_1}. By reasoning in this manner, it is easy to see that $last(I_{b_t})$ is set to a proper predecessor of e_{a_t} at the end of the tth iteration. During the $s+1$st iteration, $last(I_i)$ is set to a proper predecessor of e'. Thus, $last(I_i) = last(I_i')$ or $last(I_i) \prec last(I_i')$, a contradiction.

Case ii): $last(I_i) \prec last(I_i')$. This case cannot occur for the following reason: Clearly, at the end of an iteration, events of I_i are removed (by setting $last(I_i)$, during an iteration, to an event that is a proper predecessor of $last(I_i)$) only if the removed events of I_i cannot be in the consistency set of any event of I_k for some process P_k. Thus, we never remove an event from I_i that is in the consistency set of at least one event of the spectrum of each process.

By a similar proof, we can show that $first(I_i) = first(I_i')$ for all P_i. ■

2) A Distributed Implementation: Initially, each process P_i has local knowledge only. In order to determine which of the events of I_i must be removed, P_i exchanges information with other processes by message-passing. We now present the details of a distributed implementation of the two phases of the algorithm executed by P_i.

Phase 1: $first(I_i)$ and $last(I_i)$ are set to $first(S_i)$ and $last(S_i)$, respectively. For each process P_j, process P_i computes $\mathrm{NSM}_{ij}^{l}, \mathrm{NMR}_{ij}^{l}, \mathrm{NSM}_{ij}^{f}$, and NMR_{ij}^{f} by examining its local space-time diagram.

Phase 2: Phase 2 consists of $n-1$ iterations. At the beginning of each iteration, process P_i sends a $nms\_last(\mathrm{NSM}_{ik}^{l})$ message and a $nmr\_first(\mathrm{NMR}_{ik}^{f})$ message to process P_k. Process P_i then waits for a nms_last and nmr_first message from each process and processes these messages as follows:

Step 1: Let $nms\_last(x)$ be a message received by P_i from P_j. If NMR_{ij}^{l}, computed locally, is greater than x (the value sent by P_j), then P_i examines its local space-time diagram and finds its latest event e such that the number of messages received by P_i from P_j till e is equal to x. Set $last(I_i)$ to e if e is a predecessor of $last(I_i)$.

Step 2: Let $nmr\_first(y)$ be a message received by P_i from P_j. If y is greater than NSM_{ij}^{f}, process P_i finds its earliest event e' such that the number of messages sent by P_i to P_j till e' is equal to y. $first(I_i)$ is set to e' if e' is a predecessor of $first(I_i)$.

Process P_i ends its current iteration after it receives and processes nms_last and nmr_first messages sent by all of the processes. It then recomputes $\mathrm{NSM}_{ij}^{l}, \mathrm{NMR}_{ij}^{l}, \mathrm{NSM}_{ij}^{f}$, and NMR_{ij}^{f} for each process P_j and begins the next iteration. Note that $first(I_i)$ and $last(I_i)$ may be updated several times during an iteration, but the values $\mathrm{NSM}_{ij}^{l}, \mathrm{NMR}_{ij}^{l}, \mathrm{NSM}_{ij}^{f}$, and NMR_{ij}^{f} (for P_j) are computed only once for each iteration. Thus, during Step 1, even if NMR_{ij}^{l} is greater than NSM_{ji}^{l} and event e is found, e may not be a predecessor of $last(I_i)$.

After $n-1$ iterations, the algorithm terminates at P_i.

Message Complexity: During each iteration, two messages are sent by a process to another process. Thus, at most $O(n^2)$ message are sent during an iteration. Since the total number of iterations is $n-1$, the message complexity is $O(n^3)$.

B. Testing POSSIBLY (Φ)

We present a methodology for evaluating POSSIBLY(Φ). This methodology is based on computing the intersection of projections. If all of the intersections at the processes are nonempty, then POSSIBLY(Φ) is true, and this is formally shown below:

Lemma 1: Let $S_1, \cdots, S_n$ be spectra on processes $P_1, \cdots, P_n$ respectively. Let $I_1, \cdots, I_n$ be spectra such that $I_i = \bigcap_{j=1}^{n} \pi(S_j \rightarrow P_i)$, $1 \leq i \leq n$. *If* $I_1, I_2, \cdots, I_n$ are all nonempty, then $(first(I_1), \cdots, first(I_n))$ is a consistent cut and $(last(I_1), \cdots, last(I_n))$ is a consistent cut.

Proof: We prove by contradiction that $(last(I_1), \cdots, last(I_n))$ is a consistent cut. Assume to the contrary. Then there exist processes P_i and P_j such that $last(I_i)$ and $last(I_j)$ are not in each other's consistency sets. There are four cases to consider.

1) $last(I_i) \prec \mathcal{FCCE}_i(last(I_j))$ and $last(I_i) \neq \mathcal{FCCE}_i(last(I_j))$.
2) $\mathcal{LCCE}_j(last(I_i)) \prec last(I_j)$ and $\mathcal{LCCE}_j(last(I_i)) \neq last(I_j)$.
3) $last(I_j) \prec \mathcal{FCCE}_j(last(I_i))$ and $last(I_j) \neq \mathcal{FCCE}_j(last(I_i))$.
4) $\mathcal{LCCE}_i(last(I_j)) \prec last(I_i)$ and $\mathcal{LCCE}_i(last(I_j)) \neq last(I_i)$.

Consider Case 1. Let $e_{i_1}^{i} = \mathcal{FCCE}_i(last(I_j))$. There are two sub-cases: (a) $e_{i_1}^{i} \in S_i$. Therefore, $last(I_i) \neq last(S_i)$. This can happen only if there is some S_k such that $last(I_i) = \mathcal{LCCE}_i(last(S_k)) \prec e_{i_1}^{i}$. (Note that if for all processes P_k $e_{i_1}^{i} \prec \mathcal{LCCE}_i(last(S_k))$, $e_{i_1}^{i} \prec last(I_i)$.) Clearly, $e_{i_1}^{i}$ is not the first event of P_i; hence, there is an arc from $e_{i_1}^{i}$ to $last(I_j)$ or a proper predecessor of $last(I_j)$ because $e_{i_1}^{i} = \mathcal{FCCE}_i(last(I_j))$. $Last(S_k)$ can not be the last event of P_k because $\mathcal{LCCE}_i(last(S_k))$ is not the last event of P_i. Therefore, there is an arc from a proper successor of $last(S_k)$ to the immediate successor event (call it $e_{i_2}^{i}$) of $last(I_i)$. Note that $e_{i_1}^{i} \neq e_{i_2}^{i}$ and $e_{i_2}^{i} \prec e_{i_1}^{i}$. These mean that $\mathcal{LCCE}_j(last(S_k))$ is a proper predecessor of $last(I_j)$ so that $last(I_j) \notin I_j$ (contradiction). (b) $e_{i_1}^{i} \notin S_i$. Once again, this means that $\mathcal{LCCE}_j(last(S_i))$ is a proper predecessor of $last(I_j)$ implying that $last(I_j) \notin I_j$ (contradiction).

Consider Case 2. This implies that $last(I_i) \prec \mathcal{FCCE}_i(last(I_j))$ and the proof of Case 1 applies.

Cases 3 and 4 are similar to cases 1 and 2 respectively.

Next, once again by contradiction we prove that $(first(I_1), \cdots, first(I_n))$ is a consistent cut. Assume to the contrary. Then there exist processes P_i and P_j such that $first(I_i)$ and $first(I_j)$ are not in each other's consistency sets. There are four cases to consider.

1) $\mathcal{LCCE}_i(first(I_j)) \prec first(I_i)$ and $\mathcal{LCCE}_i(first(I_j)) \neq first(I_i)$.
2) $first(I_j) \prec \mathcal{FCCE}_j(first(I_i))$ and $first(I_j) \neq \mathcal{FCCE}_j(first(I_i))$.
3) $\mathcal{LCCE}_j(first(I_i)) \prec first(I_j)$ and $\mathcal{LCCE}_j(first(I_i)) \neq first(I_j)$.
4) $first(I_i) \prec \mathcal{FCCE}_i(first(I_j))$ and $first(I_i) \neq \mathcal{FCCE}_i(first(I_j))$.

Consider Case 1. Let $e^i_{i_1} = \mathcal{LCCE}_i(first(I_j))$. There are two sub-cases: (a) $e^i_{i_1} \in S_i$. Therefore, $first(S_i) \neq first(I_i)$. This can happen only if there is some S_k such that $first(I_i) = \mathcal{FCCE}_i(first(S_k))$. Since $first(I_i)$ is not the first event of P_i, there is an arc from $first(I_i)$ to $first(S_k)$ or a proper predecessor of $first(S_k)$. Since $e^i_{i_1}$ is not the last event of P_i, there must be an arc from a proper successor of $first(I_j)$ to the immediate successor $e^i_{i_2}$ of $e^i_{i_1}$ on P_i. We can then deduce that $e^i_{i_2} \neq first(I_i)$ and $e^i_{i_2} \prec first(I_i)$. Then, $first(I_j)$ is a proper predecessor of $\mathcal{FCCE}_j(first(S_k))$, so $first(I_j) \notin I_j$ (contradiction). (b) $e^i_{i_1} \notin S_i$. Then, $first(I_j)$ is a proper predecessor of $\mathcal{FCCE}_j(first(S_i))$ meaning that $first(I_j) \notin I_j$ (contradiction).

Consider Case 2. This implies that $\mathcal{LCCE}_i(first(I_j)) \prec first(I_i)$ and the proof of Case 1 applies.

Cases 3 and 4 are similar to cases 1 and 2 respectively. ■

Clearly, $(first(I_1), first(I_2), \cdots, first(I_n))$ denotes the consistent cut where the conjuncts $C_1, C_2, \cdots, C_n$ are all true simultaneously and the global predicate Φ is true. Thus, by Lemma 1, if the spectra $S_1, S_2, \cdots, S_n$ on processes $P_1, P_2, \cdots, P_n$ respectively, are such that $I_i = \bigcap_{j=1}^n \pi(S_j \rightarrow P_i)$ is nonempty for each process P_i, then POSSIBLY(Φ) is true. This is the main idea of the algorithm for determining POSSIBLY(Φ) that is presented next.

An Algorithm for a Single Spectrum at Each Process: Assume that the complete space-time diagram is available and each process has exactly one spectrum. Thus, when C_i is evaluated at each event of process P_i, the evaluation of each C_i results in exactly one spectrum S_i. If Φ does not contain a conjunct for some process P_j, take $first(S_j)$ = first event of P_j and $last(S_j)$ = last event of P_j. (In this case, S_j consists of all events of P_j.) At each site i, evaluate $I_i = \bigcap_{j=1}^n \pi(S_j \rightarrow P_i)$. If I_i is empty for at least at one process P_i, then POSSIBLY(Φ) is false. Otherwise, POSSIBLY(Φ) is true.

We next consider the more general case.

An Algorithm for Multiple Spectra: In general, evaluation of a conjunct C_i on process P_i will give raise to multiple spectra where C_i holds. This situation can be handled by an extension of the technique just described.

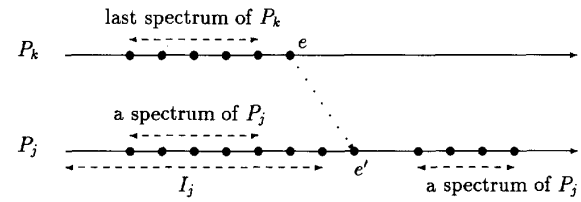


Fig. 8. A situation where a spectrum of a process may be deleted.

Testing POSSIBLY(Φ) involves selecting spectra $S_1, \cdots, S_n$ on processes $P_1, \cdots, P_n$ such that $I_i = \bigcap_{j=1}^n \pi(S_j \rightarrow P_i)$ is nonempty for each process P_i. A naive approach is to consider all combinations of selecting n spectra on n processes and, for each combination selected, finding the intersections of projections at all processes and checking if the intersections are nonempty. If $m_1, \cdots, m_n$ are the number of spectra of processes $P_1, \cdots, P_n$ respectively, then this approach tests $m_1 \times \cdots \times m_n$ combinations. We next present a technique that considers at most $m_1 + \cdots + m_n$ combinations.

1) A Sequential Algorithm: The Main Idea: Let $\Sigma = \langle e_1, e_2, \cdots, e_n \rangle$ be the latest consistent cut such that for each $1 \leq i \leq n$, e_i (an event of P_i) does not appear after the last event of the last spectrum of P_i. Thus, any consistent cut reachable from Σ (i.e., any successor of Σ) has at least one event e'_i of a process P_i such that e'_i is a proper successor of all events of all of spectra of P_i. Σ can be found as follows. For each process P_i, let CS_i, the "current spectrum" of P_i, be the sequence of all events of P_i from the first event of P_i up to and including the last event of the last spectrum of P_i. Thus, $first(\mathrm{CS}_i)$ is the first event of P_i and $last(\mathrm{CS}_i)$ is the last event of the last spectrum of P_i. Find $I_i = \bigcap_{j=1}^n \pi(\mathrm{CS}_j \rightarrow P_i)$ for each P_i. Since all "current spectra" include the first event of all processes, $I_1, I_2, \cdots, I_n$ are all nonempty. By Lemma 1, $\langle last(I_1), \cdots, last(I_n) \rangle$ is a consistent cut and is equal to Σ. For each $1 \leq i \leq n$, if $last(I_i)$ is an event of a spectrum of P_i, then Σ is a consistent cut where Φ holds, and we terminate. Otherwise, there exists at least one process P_j such that $last(I_j)$ is not an event of any spectrum of P_j. Clearly, there exists a process P_k such that $\pi(\mathrm{CS}_k \rightarrow P_j) = I_j$. (Note that the "current spectrum" of each process starts with the first event of the process.) Thus, there exist events e of P_k and e' of P_j such that i) $e \prec e'$, ii) e happens after the last event of the last spectrum of P_k, and iii) e' happens immediately after event $last(I_j)$. (See Fig. 8.) Each event of P_j that occurs after e' cannot be in the consistency set of any event of any spectrum of P_k. Thus, all spectra of P_j after e' need not be considered and can be deleted. Set CS_j to the sequence of events starting with P_j's first event and ending with the last event of the last undeleted spectrum of P_j. If P_j has no spectrum under consideration, we stop and announce POSSIBLY(Φ) to be false.

We next present the details of a centralized algorithm, which consists of two phases.

Phase 1: The first phase is the initialization phase where the "current spectrum," CS, of each process is set to all events

```
procedure Possibly;
(* algorithm executed by process P_i *)
begin
Phase 1 (* initialization *)
    at each process P_i, evaluate C_i to create 0 or more spectra;
    (* if some process has no spectrum then POSSIBLY(Φ) is false and stop *)
    first(CS_i) ← first event of P_i;
    last(CS_i) ← last event of the last spectrum of P_i;
    POSSIBLY ← undecided;
Phase 2
    while POSSIBLY = undecided loop
    Step 1
        I_i ← ⋂_{j=1}^{n} π(CS_j → P_i);
    Step 2
        if last(I_i) is an event that belongs to a spectrum of P_i then
            FOUND_i ← true;
            last(CS_i) ← last(I_i);
        else
            FOUND_i ← false;
            if P_i has no spectrum preceding last(I_i) then
                POSSIBLY ← false;
            else last(CS_i) ← last(latest spectrum of P_i that precedes last(I_i));
            endif;
        endif;
    Step 3
        if FOUND_1, FOUND_2, ..., FOUND_n are all true then
            POSSIBLY ← true;
        endif;
    endloop; (* end of while loop *)
end;
```

Fig. 9. Algorithm *Possibly* for Determining POSSIBLY(Φ).

from the beginning till the last event of its last spectrum. CS_i includes all spectra of P_i. Spectra of P_i are removed from consideration by "shrinking" CS_i.

Phase 2: Phase 2 consists of several iterations. During each iteration, if a consistent cut satisfying Φ is found, the algorithm terminates by announcing POSSIBLY(Φ) to be true. Otherwise, at least one spectrum of a process is deleted from its current spectrum CS and we proceed to the next iteration. If all the spectra of a process are deleted, the algorithm terminates and announces POSSIBLY(Φ) to be false. Each iteration of Phase 2 consists of three steps.

Step 1: Compute $I_i = \bigcap_{j=1}^{n} \pi(CS_j \rightarrow P_i)$ for each P_i using the algorithm of Section IV-A.

Step 2: If $last(I_i)$ is an event of a spectrum of P_i, then we have found a consistent cut with one event of that component in a spectrum. Thus, we set $FOUND_i$ to be true and update CS_i to include all events of P_i from the first event till $last(I_i)$.

If $last(I_i)$ is not an event of a spectrum of P_i, then we check if P_i has any spectrum that precedes $last(I_i)$. If so, we set CS_i to include all events of P_i till the last event of the last spectrum of P_i that precedes $last(I_i)$. If P_i has no spectrum that precedes $last(I_i)$, clearly all spectra of P_i have been deleted, and we stop by announcing POSSIBLY(Φ) to be false.

Step 3: If $last(I_1), last(I_2), \cdots, last(I_n)$ are all events of a spectrum of the processes $P_1, P_2, \cdots, P_n$, respectively, then the consistent cut $\langle last(I_1), last(I_2), \cdots, last(I_n)\rangle$ satisfies Φ, and we stop and announce POSSIBLY(Φ) to be true. Otherwise, the next iteration begins.

A formal description of the algorithm is given in Fig. 9, which is now shown to be correct.

Theorem 2: Algorithm Possibly *announces* POSSIBLY(Φ) *to be true iff* POSSIBLY(Φ) *is true for the space-time diagram.*

Proof: Since at least one spectrum of a process is removed during each iteration, after a finite number of iterations, the algorithm terminates. (The algorithm terminates earlier if it announces POSSIBLY(Φ) to be true.)

If part: Algorithm *Possibly* terminates and announces POSSIBLY(Φ) to be true, only when it finds that $last(I_1), \cdots, last(I_n)$ are all events of a spectrum of the processes $P_1, P_2, \cdots, P_n$, respectively. By Lemma 1, $\langle last(I_1), \cdots, last(I_n)\rangle$ is a consistent cut and Φ holds in that consistent cut. Thus, POSSIBLY(Φ) is indeed true if the algorithm announces so.

Only if part: We next show that if POSSIBLY(Φ) is true, then algorithm *Possibly* announces POSSIBLY to be true.

Let $\Sigma = \langle e_1, e_2, \cdots, e_n\rangle$ be a consistent cut such that Φ is true in Σ. Clearly, e_i is an event of a spectrum of P_i for all $1 \leq i \leq n$. Assume for contradiction that algorithm *Possibly* halts and announces POSSIBLY(Φ) to be false. Let $S = \{S_1, S_2, \cdots S_n\}$ be the set of spectra such that e_i is in S_i and S_i is a spectrum of P_i for all $1 \leq i \leq n$. To announce POSSIBLY(Φ) to be false, the algorithm must delete all spectra of a process before terminating. Let t be the first iteration of the while loop during which a spectrum of S is deleted. Clearly, at the beginning of the tth iteration, S_i is included in CS_i for all $1 \leq i \leq n$.

Consider the tth iteration. Let P_k be a process such that $S_k \in S$ is deleted. Since S_k is deleted, there exists a process P_j such that $S' = \pi(CS_j \rightarrow P_k)$ does not include any event of S_k. Clearly, no event of CS_j can be in the consistency set of any event of P_k that happens after $last(S')$. Since spectrum S_j (that includes e_i) is included in CS_j at the beginning of the tth iteration, no event of S_j can be in the consistency set of any event of S_k. (Recall that S' does not include S_k.) Thus, e_j and e_k are not in the consistency sets of each other, contradicting the assumption that $\Sigma = \langle e_1, \cdots, e_n\rangle$ is a consistent cut. ■

2) A Distributed Algorithm: The distributed algorithm is a straightforward implementation of each step of the centralized algorithm in a distributed environment. Phase 1 is performed at each process locally. When a process has no spectrum, it broadcasts a *terminate(POSSIBLY=false)* message, and all processes terminate the distributed algorithm on receiving a *terminate* message.

Phase 2 consists of several iterations. In Step 1, all processes run the algorithm of Section IV-A2). Step 2 can be performed by each process locally without exchanging any message with the other processes. During Step 2, if a process deletes all of its spectra, it broadcasts a *terminate* (*POSSIBLY=false*) message and terminates the algorithm. In Step 3, each process broadcasts its local value of FOUND and waits for the values broadcast by all of the processes. If all of the FOUND values are true, then each process terminates with POSSIBLY(Φ) to be true. Otherwise, the processes proceed to the next iteration.

Message Complexity: Clearly, the total number of iterations is at most m where m is the total number of spectra in all of the processes. During each iteration, $I_1, I_2, \cdots, I_n$ are found using the algorithm of Section IV-A2). This involves $O(n^3)$ messages. During each iteration, a process broadcasts at most two messages—a message containing its FOUND value and a *terminate* message if it has no spectrum to consider. Thus, $O(n^3)$ messages are sufficient for one iteration and $O(mn^3)$ messages are sufficient for the algorithm.

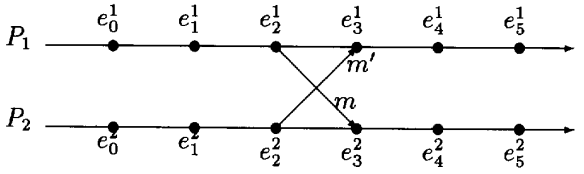


Fig. 10. Illustration of DEFINITELY(Φ).

C. Testing ALWAYS (Φ)

ALWAYS(Φ) can be evaluated by checking whether C_i holds at all events of P_i for all processes P_i. If C_i is false after some event e_j^i at some process P_i, Φ would be false in all consistent cuts that have e_j^i as a component causing ALWAYS(Φ) to be false; otherwise, ALWAYS(Φ) would be true.

D. Testing LAST (Φ)

Observe that if there is a consistent cut in which Φ is true, the algorithm of Section IV-B2) for POSSIBLY(Φ) will not only find that consistent cut but the consistent cut so found is also LAST(Φ).

E. Testing FIRST (Φ)

The algorithm for POSSIBLY(Φ) can be quite easily adapted to determine FIRST(Φ). To do this, the spectra at each process are found as in LAST(Φ). However, the spectra are considered and eliminated in the reverse order. That is, the current spectra of each process includes all events from some spectrum to the last event of that process. Thus, $last(\mathrm{CS}_i)$ is set to the last event of P_i and is not changed. Initially, $first(\mathrm{CS}_i)$ is set to the first event of the first spectrum of P_i and we compute $I_i = \bigcap_{j=1}^{n} \pi(CS_j \rightarrow P_i)$. We then eliminate spectra from the beginning instead of deleting from the end (as was done in Section IV-B2).

F. Testing DEFINITELY (Φ)

Consider a CFP Φ on a system with two processes P_1 and P_2. Let P_1 satisfy C_1 during the spectrum $S_1 = \langle e_1^1, e_2^1, e_3^1, e_4^1 \rangle$ and P_2 satisfy C_2 during the spectrum $S_2 = \langle e_1^2, e_2^2, e_3^2, e_4^2 \rangle$. The scenario is shown in Fig. 10. Message m is sent by P_1 during e_2^1 and is received by P_2 during e_3^2. Similarly, message m' is sent by P_2 during e_2^2 and is received by P_1 during e_3^1. Obviously, when P_2 completes e_3^2, the message m would already have arrived from P_1, and hence P_1 must have completed event e_2^1. Also obvious from the figure is that P_1 can not have completed e_3^1 before P_2 completes e_2^2. It follows that one of the two consistent cuts (e_2^1, e_3^2) or (e_3^1, e_2^2) must occur in any execution. Thus, Φ must hold in all possible executions of the space-time diagram, and DEFINITELY(Φ) is true.

On the other hand, if m is omitted from the figure (i.e. e_2^1 does not correspond to sending of a message to P_2 and e_3^2 does not correspond to receiving a message from P_1), no event of P_2 is dependent on any event in P_1, so an execution in which P_2 completes all of its events before P_1 even enters the spectrum S_1 is possible, which makes DEFINITELY(Φ) false.

The above scenario is an example of the following general result.

Lemma 2: Let the evaluation of Φ at each process result in exactly one nonempty spectrum S_k at each process P_k. Then, DEFINITELY(Φ) is false if and only if there exists a pair of processes P_i and P_j such that $first(S_j) \not\prec last(S_i)$ and $last(S_i)$ is not the last event of P_i.

Proof: Assume there are two processes P_i and P_j such that $first(S_j) \not\prec last(S_i)$ and $last(S_i)$ is not the last event of P_i. Obviously, $e_l^j \not\prec last(S_i)$ for any event e_l^j in S_j. Therefore, for any event $e_k^i \prec last(S_i)$, $e_l^j \not\prec e_k^i$. Then an execution is possible where P_j begins $first(S_j)$ after P_i (completes all of the events of S_i and) begins the event immediately after $last(S_i)$. Thus we have constructed an execution where Φ does not hold resulting in DEFINITELY(Φ) to be false.

To prove necessity, assume that DEFINITELY(Φ) is false. Therefore, there exists an execution $\mathcal{E}$ in which Φ evaluates to false for all consistent cuts of $\mathcal{E}$. Assume that for all pairs of processes P_i and P_j, $first(S_j) \prec last(S_i)$ or $last(S_i)$ is the last event of P_i. In $\mathcal{E}$, let Σ_1 be the first consistent cut where each process P_k has executed $first(S_k)$, and let Σ_2 be the first consistent cut where some process P_l has executed the immediate successor of $last(S_l)$. (Clearly, $last(S_l)$ is not the last event of P_l and such a process P_l exists since DEFINITELY(Φ) is false.) If $\Sigma_1 \prec \Sigma_2$, consider the consistent cut $\Sigma_3 \prec \Sigma_2$, such that P_l has just executed $last(S_l)$. Since $last(S_l)$ is not the last event of P_l, $first(S_j) \prec last(S_l)$ for each process P_j and P_j must have entered S_j in Σ_3. Because $\Sigma_3 \prec \Sigma_2$, each P_j cannot have left S_j. Thus, in Σ_3 all processes must be in their respective spectra, and Φ must hold in $\mathcal{E}$ (contradiction). So $\Sigma_2 \prec \Sigma_1$ in $\mathcal{E}$. Therefore, P_l can complete $last(S_l)$ before some process P_k executes $first(S_k)$, proving that $first(S_k) \not\prec last(S_l)$. ■

Corollary 1: DEFINITELY(Φ) is true if and only if there exist spectra $S_1, S_2, \cdots, S_n$ *on processes* $P_1, P_2, \cdots, P_n$, respectively, such that for each ordered pair of processes P_i, P_j, $first(S_j) \prec last(S_i)$ or $last(S_i)$ is the last event of P_i.

If for some process P_i the evaluation of C_i at P_i results in no spectrum, DEFINITELY(Φ) is obviously false. Thus, if the complete space-time diagram is available and for all processes P_i, the evaluation of C_i at process P_i results in at most one spectrum S_i, the above lemma can be used to test DEFINITELY(Φ).

A More General Case: A naive approach to handle the more general case where one or more process may have multiple spectra is to consider every possible combination of spectra by choosing exactly one spectrum from each process and apply the test suggested by Lemma 2. However, such a test could take a very long time to complete because the number of tests may be exponential in the number of processes. We first present a sequential algorithm for detecting DEFINITELY(Φ) efficiently. A distributed implementation is given in Section IV-F2).

The Main Idea: For each process P_i, exactly one spectrum is selected as its *current spectrum*, denoted by CS_i. We say that *property A holds for* P_i *with respect to* P_j if $first(\mathrm{CS}_j) \prec last(\mathrm{CS}_i)$. We maintain a set C of processes such that if $P_i \in C$, then either $last(\mathrm{CS}_i)$ is the last event of P_i or property

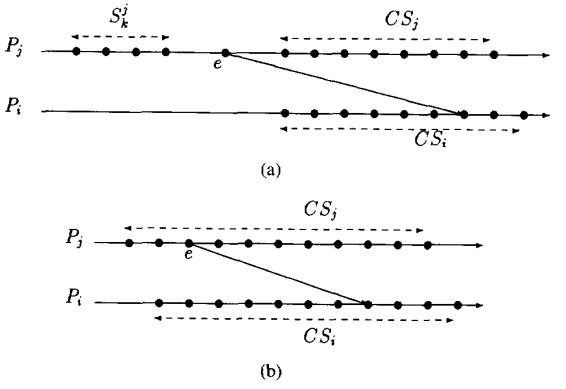


Fig. 11. How to change the current spectrum. Case (a). Case (b).

A holds for P_i with respect to each process. Clearly, for each (ordered) pair of processes P_i and P_j in $\mathcal{C}$, if $last(\mathrm{CS}_j)$ is not the last event of P_j, $first(\mathrm{CS}_i) \prec last(\mathrm{CS}_j)$. By Corollary 1, if $\mathcal{C}$ includes all processes, DEFINITELY(Φ) is true. Initially, for each process P_i, CS_i is the last spectrum of P_i and $\mathcal{C}$ contains all processes P_j such that $last(\mathrm{CS}_j)$ is the last event of P_j. ($\mathcal{C}$ is ϕ if no such process exists.) Iteratively choose some process P_k not in $\mathcal{C}$ and add P_k to $\mathcal{C}$. When adding P_k to $\mathcal{C}$, adjust the current spectrum of all other processes such that property A holds for P_k with respect to all other processes.

Let $\mathrm{LPE}(P_j, last(\mathrm{CS}_i))$ be the latest event of P_j that is a predecessor of $last(\mathrm{CS}_i)$. Pick any process, say $P_i \notin \mathcal{C}$. For every process $P_j \neq P_i$, determine $\mathrm{LPE}(P_j, last(\mathrm{CS}_i))$ and call it e. Depending on the occurrence of e within P_j, different courses of actions are taken to satisfy property A:

1) If e occurs before $first(\mathrm{CS}_j)$ (see Fig. 11, case (a)[1]), $first(\mathrm{CS}_j)$ is not a predecessor of any event of the current spectrum of P_i. (If it were, then it would also be a predecessor of $last(\mathrm{CS}_i)$.) As a consequence, CS_j can be discarded from further consideration, and the current spectrum of P_j is set to the latest spectrum of P_j among all spectra of P_j that precede e or include e. Thus, CS_j is adjusted in the "backward direction" for each process P_j. When CS_j is adjusted, we check if P_j is in $\mathcal{C}$. Clearly, if $P_j \in \mathcal{C}$, then property A may not hold for P_j with respect to some process $P_k \neq P_j$ after CS_j is adjusted. Thus, if $P_j \in \mathcal{C}$ and CS_j is adjusted, we remove P_j from $\mathcal{C}$ so that property A holds at all times for each process in $\mathcal{C}$ with respect to all processes.
2) The second case is similar to case (1) above, but with one difference: there is no spectrum that includes e and there is no spectrum preceding event e. Clearly, there is no more spectrum from P_j to consider. In this case, conclude that DEFINITELY(Φ) is false and halt the algorithm.
3) If e is an event in CS_j (the current spectrum of P_j), no action is warranted since property A holds for P_i with respect to P_j. (See Fig. 11, case (b).)

[1]Recall that initially CS_i and CS_j are the last spectra on P_i and P_j respectively.

```
procedure definitely;
begin
Phase 1 (* initialization *)
    at each process P_i, evaluate C_i to create 0 or more spectra;
    (* if some process has no spectrum then DEFINITELY(Φ) is false and stop *)
    DEFINITELY ← undecided;
    CS_i ← last spectrum on P_i for each process P_i;
    C ← {P_j | last(CS_j) is the last event of P_j};
Phase 2
    while DEFINITELY = undecided loop
    Step 1
        choose a process P_i ∉ C;
    Step 2
        for each process P_j ≠ P_i loop
        Step 2.1
            e ← LPE(P_j, last(CS_i));
        Step 2.2
            if e occurs before first(CS_j) then
                if there exists a spectrum S_j such that first(S_j) = e or first(S_j) ≺ e then
                    CS_j ← latest such spectrum;
                    C ← C − { P_j }; (* no change in C if P_j ∉ C *)
                else DEFINITELY ← false;
                endif;
            endif;
        endloop;
    Step 3
        C ← C ∪ {P_i};
        if C includes all participating processes then DEFINITELY ← true;
        endif;
    endloop; (* end of while loop *)
end;
```

Fig. 12. Centralized Algorithm for Determining DEFINITELY(Φ).

1) A Sequential Algorithm: The algorithm consists of two phases.

Phase 1: Phase 1 is the initialization phase. For each process P_i, its conjunct C_i is evaluated to create 0 or more spectra. If Φ does not involve conjunct C_j, (i.e., the predicate does not involve any variable local to P_j), process P_j is removed from consideration, and P_j does not participate in the algorithm. For each participating process P_i, CS_i is set to the last spectrum of P_i. $\mathcal{C}$ is initialized to $\{P_j \mid last(\mathrm{CS}_j)$ is the last event of $P_j\}$ and DEFINITELY is set to *undecided.*

Phase 2: The second phase proceeds in several rounds and each round consists of three steps—Step 1, Step 2, and Step 3.

Step 1: In step 1, a process P_i that is not in $\mathcal{C}$ is chosen.

Step 2: Step 2 consists of $n-1$ iterations and each iteration consists of Step 2.1 and Step 2.2. During each iteration of Step 2, a process $P_j \neq P_i$ is chosen. (Recall that P_i is the process chosen in Step 1.)

Step 2.1: $\mathrm{LPE}(P_j, last(CS_i))$, the latest event of P_j that is a predecessor of $last(\mathrm{CS}_i)$, is found. Let e be the event.

Step 2.2: If event e of P_j occurs before $first(\mathrm{CS}_j)$, delete the current spectrum CS_j from consideration. Let S_j be the latest spectrum of P_j such that S_j includes e or $first(S_j) \prec e$. Now, CS_j is set to S_j. As explained earlier, when CS_j is updated, P_j is removed from $\mathcal{C}$. If no such event e exists (that is, $\mathrm{LPE}(P_j, last(\mathrm{CS}_i))$ does not exist) or e occurs before the first event of the first spectrum of P_j, we halt and announce DEFINITELY(Φ) to be false.

Step 3: In Step 3, process P_i chosen in Step 1 is added to $\mathcal{C}$. If $\mathcal{C}$ includes all of the participating processes, then we halt and announce DEFINITELY(Φ) to be true.

A formal description of the algorithm appears in Fig. 12. Each round of the algorithm corresponds to one iteration of the while loop of Fig. 12.

Theorem 3: The algorithm in Fig. 12 correctly computes DEFINITELY(Φ). That is, the algorithm announces "true" if and only if DEFINITELY(Φ) is true.

Proof: We first show that if the algorithm announces "true," then DEFINITELY(Φ) is indeed true. For this, first observe that this announcement occurs only if the variable DEFINITELY is true, which happens only if $\mathcal{C}$ includes all participating processes. Thus, if we show that for all (ordered) pairs of processes P_i and P_j in $\mathcal{C}$, $first(\mathrm{CS}_i) \prec last(\mathrm{CS}_j)$ or $last(\mathrm{CS}_j)$ is the last event of P_j, then by Corollary 1 the result follows. Since $\mathcal{C}$ is initialized properly, this is equivalent to showing that for each P_i that was added to $\mathcal{C}$ in Step 3, property A holds for P_i with respect to all other processes.[2] This proof is by induction on the number of rounds (iterations of the while loop in Fig. 12).

For induction basis, initially $\mathcal{C}$ does not have any process that was added in Step 3. For induction hypothesis, assume that for each $P_j \in \mathcal{C}$, property A holds for P_j with respect to all other processes at the end of the first $t \geq 0$ rounds. In the $t+1$st round, only the following four actions are performed that affects property A: a) we choose some process P_i not in $\mathcal{C}$, b) determine $\mathrm{LPE}(P_j, last(\mathrm{CS}_i))$ for each process $P_j \neq P_i$, c) update CS_j for each P_j based on $\mathrm{LPE}(P_j, last(\mathrm{CS}_i))$, and d) include P_i in $\mathcal{C}$. Clearly actions a) and b) do not affect property A. Consider action c) for each process P_j where CS_j is updated. (If CS_k is not updated during a round, then it is easy to see that property A holds.) If $P_j \in \mathcal{C}$ and CS_j is updated, then P_j is deleted from $\mathcal{C}$. In this case, property A holds. Assume that $P_j \notin \mathcal{C}$. Since CS_j is updated in the "backward direction" (i.e., CS_j at the end of the $t+1$st round is a proper predecessor spectrum of CS_j at the end of the tth round), at the end of the $t+1$st round, $first(\mathrm{CS}_j) \prec last(\mathrm{CS}_i)$ for each P_j if P_i was in $\mathcal{C}$ at the end of the tth round and P_i is not deleted from $\mathcal{C}$ in the $t+1$st round. Thus, property A holds after action c). Finally, action d) preserves property A. Therefore, property A holds at the end of the $t+1$st round.

To prove sufficiency, assume DEFINITELY(Φ) is true. Then, there exists at least one set of spectra $\mathcal{S} = \{S_1, \cdots, S_n\}$ one spectrum per process satisfying Corollary 1. Choose $\mathcal{S}$ so that Corollary 1 does not hold for any other set $\{S'_1, \cdots, S'_n\}$ where S'_i is a successor of S_i for $1 \leq i \leq n$. (Thus, we are choosing the latest possible combination of spectra.) Clearly, for each (ordered) pair of processes P_i, P_j, $first(S_i) \prec last(S_j)$ or $last(S_j)$ is the last event of P_j.

We show by induction on the number of iterations of the while loop that the algorithm will not allow the current spectrum of a process P_j to be a proper predecessor of S_j. For induction basis, before the first iteration, the current spectrum of a process is the last spectrum, and the claim holds. Assume the claim holds after t iterations. Consider the $t+1$st iteration where we choose P_i (note that $P_i \notin \mathcal{C}$) and determine for all other processes P_j, $\mathrm{LPE}(P_j, last(CS_i))$. Since $\mathrm{CS}_i = S_i$ or CS_i is a successor of S_i at the end of the tth round and $first(S_j) \prec last(S_i)$ for all processes P_j, no process P_j will change CS_j to a spectrum that is a proper predecessor of S_j, so the claim holds at the end of the $t+1$st iteration as well.

Due to the claim we just proved, if DEFINITELY(Φ) is true, no process will ever be left without a current spectrum, so we can be sure that the algorithm will not announce "false." Then, to complete the proof for sufficiency, we only need to show that the algorithm will eventually terminate when DEFINITELY(Φ) is true. Consider the while loop in the algorithm in Fig. 12. During each execution of the loop, $\mathcal{C}$ undergoes a change—one process gets added and zero or more processes are removed. A process is removed from $\mathcal{C}$ only because it changes its current spectrum, which means we are discarding one more spectrum from further consideration. Let m be the total number of spectra among the n processes. The n processes can not leave $\mathcal{C}$ more than m times. A process can be added to $\mathcal{C}$ only after it is removed from $\mathcal{C}$ (except for the first time it is added to $\mathcal{C}$). Thus, the while loop can not execute more than $m+n$ times and the algorithm terminates after at most $m+n$ executions of the while loop. ■

2) A Distributed Algorithm for DEFINITELY(Φ): We now extend the sequential algorithm for computing DEFINITELY(Φ) to a distributed environment where a process has only its local space-time diagram and message-passing is used for exchanging information. For each action of the sequential algorithm of Section IV-F1), we present the details of a distributed implementation. The set $\mathcal{C}$ is stored distributively by using a local variable IN_C at each process such that $P_i \in \mathcal{C}$ if and only if the variable IN_C at P_i is true. The details of the distributed algorithm executed by process P_i is given below.

Phase 1: At process P_i, CS_i is set to its last spectrum. IN_C is set to true if $last(\mathrm{CS}_i)$ is P_i's last event and to false otherwise.

Phase 2: Phase two consists of several rounds and each round consists of three steps. Each round of phase two is initiated by one process that is not in $\mathcal{C}$. In Step 1 of each round of Phase 2, a leader is elected among those processes whose IN_C value is false.

Step 1: Several algorithms exist in the literature for leader election. Since selecting one process among those processes not in $\mathcal{C}$ is performed at each round, the following simplified scheme for election may be used.

Assume that a spanning tree rooted at the process with the largest process id exists. (This can be constructed once for the entire algorithm as a preprocessing step.) Using the spanning tree, the function maximum is computed among the local values of the processes where a process that does not belong to $\mathcal{C}$ has its process id as its local value; a process that belongs to $\mathcal{C}$ has a null value as its local value. (A null value is lower than the process id of all processes.) The widely-used convergecast method [22] may be used for computing the maximum. If the maximum value is equal to null, all processes are in $\mathcal{C}$, the process with the largest id broadcasts a *terminate* (DEFINITELY=*true*) message to all processes, and every process terminates the algorithm on receiving a *terminate* message. If the maximum value is not equal to null, the process whose id is equal to the maximum computed value becomes the leader of the current round.

If P_i is the leader of the current round, then it broadcasts an *initiate_step_2* message to all processes (including itself).

Step 2: Unlike the sequential version, which finds LPE in iterations (one interaction for one process of the system), we coordinate all of the processes, and all processes find their

[2] Recall that property A holds for P_i with respect to P_j if $first(\mathrm{CS}_j) \prec last(\mathrm{CS}_i)$.

respective LPE events in cooperation with the others. Step 2 at process P_i begins on receipt of the *initiate_step_2* message. For each P_i construct a spectrum S_i.

If P_j is the leader of the current round, S_j consists of exactly one event—the last event of the current spectrum, CS_j, of P_j.

If process P_i is not the leader, let e_i' be a fictitious event of P_i that precedes all other events of P_i and let S_i include all events of P_i (including the fictitious event e_i'). (Thus, $first(S_i) = e_i'$ and $last(S_i)$ is the last event of P_i.)

Step 2.1: Let $I_i = \bigcap_{t=1}^{n} \pi(S_t \rightarrow P_i)$. Evaluating the intersection of the projections at all of the processes can be accomplished by using the distributed algorithm of Section IV-A2).

If P_i is not the leader, check if $first(I_i)$ is equal to e_i'. If so, there is no event of P_i that precedes the last event of the current spectrum of the leader of the current round and P_i halts after broadcasting a *terminate* (DEFINITELY= *false*). If $first(I_i) \neq e_i'$, then $e = first(I_i)$.

Step 2.2: Step 2.2 is executed by P_i only if P_i is not the leader. (If P_i is the leader, then P_i skips Step 2.2 and executes Step 3.) If e is an event that occurs before $first(\mathrm{CS}_i)$, process P_i sets IN_C to false and P_i changes its current spectrum. If e occurs before $first(\mathrm{CS}_i)$, then the latest spectrum of P_i that includes e or a predecessor of e is chosen as CS_i. If no such spectrum exists, P_i broadcasts a *terminate*(DEFINITELY=*false*) message, and P_i terminates.

Step 3: If P_i is the leader, it sets IN_C to true, terminates the current round, and sends a message to the process with the largest id informing the completion of Step 3.

Process P_i terminates the algorithm on receiving a *terminate* message.

Message Complexity: Phase 1 does not involve sending of any message. The maximum number of rounds may be $m + n$ where m is the total number of spectra on all of the processes. Step 1 involves choosing a leader and this can be done using $O(n)$ messages assuming that a spanning tree is constructed in a preprocessing step using $O(e' + n \log n)$ messages where e' is the total number of communication links. This preprocessing step is performed only once. During a round, one *initiate_step_2* message is broadcast using $O(n)$ messages and the algorithm of Section IV-A2) that uses $O(n^3)$ messages is invoked once. The algorithm terminates when a process broadcasts a *terminate* message. Thus, in a system with n processes DEFINITELY(Φ) can be computed using $O(mn^3)$ messages, where m is the total number of spectra.

V. Related Work

The works most related to this work are by Miller and Choi [20], Spezialetti [27], Spezialetti and Kearns [28], Cooper and Marzullo [8], Marzullo and Neiger [18], Manabe and Imase [17] and Garg and Waldecker [11]. Miller and Choi [20] consider linked predicates in debugging. The concepts of POSSIBLY(Φ), DEFINITELY(Φ), FIRST(Φ) and LAST(Φ) are from Cooper and Marzullo [8], and POSSIBLY(Φ) is similar to the *event occurrence* condition of [27].

The algorithms of Cooper and Marzullo [8] and Marzullo and Neiger [18] test predicates that are more general than CFP's by explicitly constructing and testing each possible consistent cut. Their algorithms are centralized and may need an exponential amount of time (and an exponential number of messages if a distributed implementation is attempted).

The technique of Manabe and Imase [17] for POSSIBLY(Φ) uses two identical runs of the distributed program being debugged—in the first run, the relative order of events is recorded (for reproducible execution) and in the second run (where the execution of each process is identical to its run in the first run) the progress within each process is carefully controlled so as to produce a consistent cut in which Φ holds. This technique works for arbitrary Φ, but two identical runs of the distributed algorithm being debugged are necessary.

Independent of us, Garg and Waldecker [11] consider POSSIBLY(Φ) and DEFINITELY(Φ) (called weak conjunctive predicates and strong conjunctive predicates in [11]) where Φ is restricted to conjunctive predicates with each conjunct involving variables of no more than one process. Their algorithms for POSSIBLY and DEFINITELY [11] are centralized and use vector clocks of Fidge [10] and Mattern [19]. Each participating process sends the vector clocks of its events during which Φ is locally satisfied to a checker process. The checker process receives the clock values of the events and finds a consistent cut where Φ holds. Although an efficient implementation of vector clocks was proposed by Singhal and Kshemkalyani [26], the vector clocks involve appending $O(n)$ numbers to each message of the distributed program being tested and debugged resulting in increased load on the communication subsystem.

There are some important differences between the work of Garg and Waldecker [11] and the one reported in this paper. First, Garg and Waldecker perform an on-line evaluation of the predicates, whereas we do an off-line evaluation. Our algorithms assume that it is possible to have reproducible execution, which can be performed using techniques suggested in the literature [4], [16], [29]. Their algorithms are distributed, but not completely decentralized: they use a central debugger that collects information from all processes and evaluates the predicates. Ours are fully distributed techniques, so all processes execute the same algorithm. Since there is no centralized debugger, our methods are easily scalable. Their algorithms do not assume the links to be FIFO and hence they are less restrictive than our algorithms, which work only if the links are FIFO. Finally, our algorithms do not use vector clocks, which significantly increase the communication overhead.

As mentioned earlier in the paper, a case could be made for both on-line and off-line evaluation strategies. An on-line technique is attractive because all predicates are evaluated during the test, so repeated runs become unnecessary. This could be important in situations where the coordination of the testing activity, which may involve several geographically distant sites, may be a time-consuming process. However, a closer examination reveals a few disadvantages of the on-line strategy.

First, an on-line evaluation may not be achieved in some cases. In an on-line evaluation of FIRST(Φ), due to communication delays, it may be impossible to know that FIRST(Φ)

has occurred until the system has changed to some successor state. When LAST(Φ) is being evaluated, we do not know what LAST(Φ) is until after the end of the run.

Second, individual processes must receive additional messages to know when to freeze their activity to permit the examination of program variables, etc. Introduction of these extra messages may introduce causal dependencies that are not otherwise present. Thus, the test perturbs the computation, which must be avoided. An off-line strategy can avoid this problem because after the off-line evaluation of predicates, information can be stored and used in subsequent runs.

VI. Conclusion and Future Work

The problem of detecting global predicates efficiently is considered and solutions have been presented in this paper. Algorithms for POSSIBLY(Φ), DEFINITELY(Φ), FIRST(Φ) and LAST(Φ) have been given. These techniques analyze a given "run" (or an execution) of a distributed program. Clearly, a single run of a distributed program creates an exponential number of consistent cuts, and it is possible to detect global predicates without explicitly considering each possible consistent cut. We have considered testing a given execution for the presence of errors. This paper does not consider the problem of generating *distributed test cases* systematically. Program testing is very difficult and is undecidable even for sequential programs [13]. Testing distributed programs is harder than testing sequential programs due to the asynchrony and combinatorial explosion of state space [31]. Taylor and Kelly [30] and Carver and Tai [5] propose a program based static analysis technique for testing concurrent synchronous programs (ADA and CSP).

Systematically generating communication patterns one by one for completely testing the program (using our results) is the next step in testing distributed programs. There is no way to generate a finite number of distributed test cases which will "expose" all errors since the equivalent problem in sequential case is known to be undecidable. Different approaches such as placing restrictions on the type of distributed programs, probabilistic approaches, combining the methods used in verification with testing [15], using the techniques developed in debugging [2], and using the techniques used in protocol testing [1], [7], [23], [24] may be attempted. We are working on these and related problems.

Acknowledgment

The authors are indebted to the anonymous referees for their insightful comments, which greatly improved the presentation of the paper.

References

[1] A. Aho, A. Dahbura, D. Lee, and M. Uyar, "An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours," in *Proc. IFIP Int. Workshop on Protocol Specification, Testing and Verification,* 1988.

[2] P. Bates and J. Wiledon, "High-level debugging of distributed systems: The behavioral abstraction approach," *J. Syst. Software 3,* vol. 4, pp. 255–264, 1983.

[3] P. A. Bernstein, N. Goodman and V. Hadzilacos, *Concurrency Control and Recovery in Database Systems.* Reading, MA: Addison-Wesley, 1987.

[4] R. Carver and K. Tai, "Reproducible testing of concurrent software based on shared variable," in *the 11th IEEE Int. Conf. Distrib. Comput. Syst.,* 1991, pp. 552–559.

[5] ——, "Static analysis of concurrent software for deriving synchronization constraints," in *Proc. Eleventh IEEE Int. Conf. Distrib. Comput. Syst.,* 1991, pp. 544–551.

[6] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst. 3,* vol. 1, 1985, pp. 63–75.

[7] W.-H. Chen, C.-S. Lu, E. Brozovsky and Wang, "An optimization technique for protocol conformance testing using multiple UIO sequences," *Inform. Process. Lett.,* vol. 36, pp. 7–12, 1990.

[8] R. Cooper and K. Marzullo, "Consistent detection of global predicates," *SIGPLAN Notices,* pp. 167–174, 1991.

[9] J. Fagerstrom, "Design and test of distributed applications," in *Proc. Tenth Int. Conf. Software Eng.,* pp. 88–92, 1988.

[10] J. Fidge, "Timestamps in message passing systems that preserve the partial ordering," in *Proc. 11th Australian Comput. Sci. Conf.,* 1988, pp. 55–66.

[11] V. Garg and B. Waldecker, "Detection of unstable predicates in distributed programs," in *Proc. Int. Conf. Foundations of Software Technol. and Theoretical Comput. Sci.,* 1992, Springer-Verlag.

[12] D. Haban and W. Weigel, "Global events and global breakpoints in distributed systems," in *Proc. 21st Int. Conf. Syst. Sci.,* 1988, pp. 166–175.

[13] W. E. Howden, "Reliability of the path analysis testing strategy," *IEEE Trans. Softw. Eng.,* vol. 3, pp. 471–482, Feb. 1976.

[14] L. Lamport, "Time, clocks and the ordering of events in a distributed system," *Commun. ACM,* pp. 558-565, 1978.

[15] W. Lloyd and P. Kearns, "Using tracing to direct our reasoning about distributed programs," in *Proc. 11th IEEE Int. Conf. Distrib. Comput. Syst.,* 1991, pp. 552–559.

[16] T. Leblanc and J. Mellor-Crummy, "Debugging parallel programs with instant replay," *IEEE Trans. Comput.* , vol. C-36, pp. 471–482, 1987.

[17] Y. Manabe and M. Imase, "Global conditions in debugging distributed programs," *J. Parallel Distrib. Comput.,* pp. 62–69, 1992.

[18] K. Marzullo and G. Neiger, "Detection of globalstate predicates," in *Proc. Fifth Int. Workshop on Distrib. Algorithms,* 1991, Springer-Verlag, pp. 254–272.

[19] F. Mattern, "Virtual time and global states of distributed systems," in *Parallel and Distrib. Algorithms: Proceedings of the Int. Workshop on Parallel and Distrib. Algorithms,* 1989, pp. 215–226.

[20] B. Miller and J.-D. Choi, "Breakpoints and halting in distributed programs," in *8th IEEE Int. Conf. Distrib. Comput. Syst.,* 1988, pp. 316–323.

[21] M. Muhlhauser, "Software engineering for distributed applications: The design project," in *Proc. Tenth Int. Conf. Software Eng.,* 1988, pp. 93–101.

[22] K. Ramarao and S. Venkatesan, "On finding and updating shortest paths distributively," *J. Algorithms,* vol. 13, no. 2, pp. 235–257, 1992.

[23] K. Sabnani and A. Dahbura, "A protocol test generation procedure," *Comput. Networks and ISDN Syst.,* vol. 15, pp. 285–297, 1988.

[24] Y. Shen, F. Lombardi, and A. Dahbura, "Protocol conformance testing using multiple UIO sequences," in *Proc. IFIP Int. Workshop on Protocol Specification, Testing and Verification,* 1989.

[25] M. Singhal, "A dynamic information-structure mutual exclusion algorithm for distributed systems," *IEEE Trans. Parallel Distrib. Syst.,* vol. 3, pp. 121–125, 1992.

[26] M. Singhal and A. Kshemkalyani, "An efficient implementation of vector clocks," *Inform. Process. Lett.,* vol. 43, pp. 47–52, 1992.

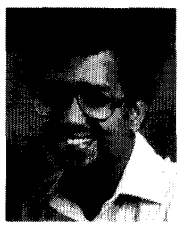
[27] M. Spezialetti, "A generalized approach to monitoring distributed computations for event occurrences," Ph.D. dissertation, Univ. Pittsburgh, Pittsburgh, PA, 1989.

[28] M. Spezialetti and P. Kearns, "Simultaneous regions: A framework for the consistent monitoring of distributed computations," in *Proc. Ninth IEEE Int. Conf. Distrib. Comput. Syst.,* 1989, pp. 61–68.

[29] K. Tai, R. Carver, and E. Obaid, "Debugging Ada concurrent programs by deterministic execution," *IEEE Trans. Software Eng.,* vol. 17, pp. 45–63, 1991.

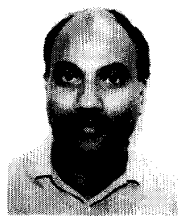
[30] R. Taylor, D. Levine and C. Kelly, "Structural testing of concurrent programs," *IEEE Trans. Softw. Eng.,* vol. 18, pp. 206–215, 1992.

[31] C. West, "Protocol validation in complex systems," in *Proc. ACM Symp. Commun. Archit. and Protocols,* 1989, pp. 303–312.



S. Venkatesan received the B.Tech. and M.Tech. degrees from the Indian Institute of Technology, Madras in 1981 and 1983, respectively, and the M.S. and Ph.D. degrees in computer science from the University of Pittsburgh in 1985 and 1988, respectively.

He joined the University of Texas at Dallas in January 1989, where he is currently an Assistant Professor of Computer Science. His research interests are in fault-tolerant distributed systems, testing and debugging distributed programs, fault-tolerant telecommunication networks, and mobile computing.



Brahma Dathan received the B.S. degree in electronics and communications engineering from University of Kerala, India, the M.S. degree in computer science from Indian Institute of Technology, Madras, and the M.S. and Ph.D. degrees in computer science from University of Pittsburgh.

He is currently with the Computer Science Department, St. Cloud State University. Prior to that, he was an Assistant Professor of Computer Science at the University of Wyoming. His research interests are in the areas of distributed systems and database systems.