Techniques to Tackle State Explosion in Global Predicate Detection

Sridhar Alagar and Subbarayan Venkatesan, *Member*, *IEEE Computer Society*

Abstract—Global predicate detection, which is an important problem in testing and debugging distributed programs, is very hard due to the combinatorial explosion of the global state space. This paper presents several techniques to tackle the state explosion problem in detecting whether an arbitrary predicate Φ is true at some consistent global state of a distributed system. We present space efficient on-line algorithms for detecting Φ. We then improve the performance of our algorithms, both in space and time, by increasing the granularity of the execution step from an event to a sequence of events in each process.

Index Terms—Distributed systems, global states, global predicates, lattice, space complexity, global intervals.

1 Introduction

Some safety properties of programs can be expressed in terms of predicates. So, a technique to detect whether a predicate becomes true in a given execution is important for testing and debugging distributed programs. In this paper, we consider the problem of detecting whether in a given distributed execution there exists a global state at which a given global predicate Φ is true. This problem was posed as, Possibly(Φ) by Cooper and Marzullo [3].

Cooper and Marzullo [3] present a centralized algorithm for detecting Possibly(Φ) for an arbitrary predicate Φ. In this paper, we refer to this algorithm as CM algorithm. The worst case space and time complexities of the CM algorithm are exponential in the number of processes. Several researchers have presented polynomial time algorithms for detecting a global predicate by placing restrictions on the type of predicates [1], [5], [9], [10], [18]. The CM algorithm is important because 1) the existing polynomial time algorithms are for restricted forms of predicates and 2) the polynomial time algorithms are different for different class of predicates. In contrast, the CM algorithm may be used to detect an arbitrary predicate. It appears that detecting an arbitrary global predicate involves exhaustive search. Alternatively, symbolic techniques may be used to detect global predicates instead of examining all reachable states [16].

A natural question to ask now is whether we can alleviate the problem of global state explosion while detecting Possibly(Φ) for an arbitrary predicate. In this paper, we present several methods to tackle state explosion while detecting Possibly(Φ). The summary of our results is as follows:

- First we present, in Section 3, a space efficient online algorithm for Possibly(Φ) that uses $O(mn)$ space, where m is the total number of events in the computation and n is the number of processes in the system. The space complexity of the algorithm is further reduced to $O(m)$ in Section 5.
- In Section 4, we further improve the performance of the algorithms by increasing the granularity of an execution step from an event to a sequence of events (*interval*). Instead of testing every global state, we test every *global interval*. When the values of the variables related to the global predicates are not changed "frequently," the number of global intervals can be substantially less than the number of global states, thereby reducing the space and time requirements of our algorithms.

Note that there are no restrictions on Φ, it can be arbitrary.

2 Preliminaries

A distributed system is a collection of n processes labeled $P_1, \ldots, P_n$. Processes are connected by point-to-point logical channels. Processes and channels are asynchronous and both are fault-free. Processes communicate by message passing only.

A process is a collection of events, which form a total order. An event in a process is an action that changes the state of the process. An event may be a *send* event resulting in sending of a message to other processes, a *receive* event resulting in receipt of a message from another process, or an *internal* event in which no sending or receiving of a message is involved. We use Lamport's partial order *happened before* [8], denoted by $\rightarrow$, to express the causality between two events. Let E be the set of all events in a particular execution. Then, $(E, \rightarrow)$ is a partially ordered set. An execution of a distributed program can be represented by a space-time diagram. The space-time diagram for a sample execution is shown in Fig. 1a.

A *consistent cut* C is a finite subset of E such that $e \in C$ and $e' \rightarrow e$ implies $e' \in C$. The *frontier* of a cut C is $\langle e_1, \ldots, e_n \rangle$ such that for all i, 1) e_i is in P_i and 2) for any e_i' in P_i, if $e_i \rightarrow e_i'$ then $e_i' \notin C$. In Fig. 1a, the frontier of the cut consisting of e_1 in P_1 and events e_1 and e_2 in P_2 is $\langle e_1, e_2 \rangle$.

• *The authors are with the Department of Computer Science, EC31, University of Texas at Dallas, Richardson, TX 75083-0688. E-mail: venky@utdallas.edu.*

Manuscript received 6 Mar. 1997; revised 5 May 1998; accepted 1 Aug. 2000. Recommended for acceptance by K. Marzullo.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 104109.

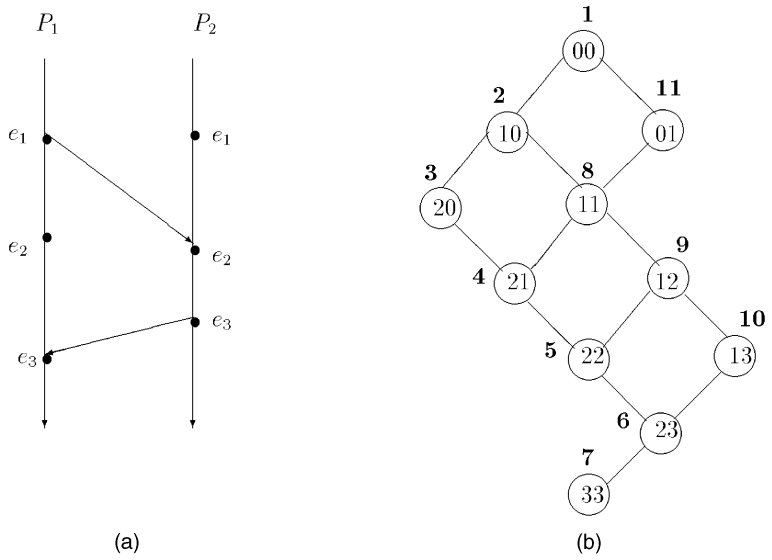

Fig. 1. (a) A sample execution. (b) The lattice for the sample execution.

A *global state* of a distributed system is a collection of local states of the processes [2]. For every consistent cut there is a corresponding consistent global state (the state of the system after the execution of all the events in the cut) and vice versa. A global state is consistent if its corresponding cut is consistent. A global state S is represented as $\langle k_1, \ldots, k_n \rangle$ such that, for process P_i, its state after executing k_i events is included in S. We say that a global state *includes* an event e if e belongs to the cut corresponding to the global state.

From now on, the term global state refers to consistent global state.

The set of all global states of an execution forms a *lattice* [12]. Two global states of a lattice are connected by an edge, if the system can proceed from one to the other by executing one event. The *initial* global state of the lattice is the global state $\langle 0, \ldots, 0 \rangle$. A path starting from the initial global state in the lattice is a *computation* of the system. The lattice for the sample execution of Fig. 1a is shown in Fig. 1b. For simplicity, a global state $\langle k_1, k_2 \rangle$ is labeled by $k_1 k_2$ in the lattice.

In a lattice, the *level* of a global state $\langle k_1, \ldots, k_n \rangle$ in a lattice is the sum $k_1 + k_2 + \ldots + k_n$. For a global state $S = \langle k_1, \ldots, k_n \rangle$,

$$value(S) \text{is the ordered } n\text{-tuple } (k_1, k_2, \ldots, k_n).$$

For a set of global states $\mathcal{SS}$,

$$value(\mathcal{SS}) = \{value(S) \mid S \in \mathcal{SS}\}.$$

For any two global states $S = \langle k_1, \ldots, k_n \rangle$ and $S' = \langle k'_1, \ldots, k'_n \rangle$, $value(S) > value(S')$ if there exists a $j \geq 1$ such that $k_j > k'_j$ and $k_i = k'_i$ for each $i \in \{1, ..., j-1\}$. The predecessor function *pred* for a global state S is defined below.

$$pred(S) = \{S' \mid \text{system may change from global state } S' \text{ to } S \text{ by executing one event}\}.$$

The successor function *succ* for a global state S is defined below.

$$succ(S) = \{S'' \mid \text{system may change from global state S to } S'' \text{ by executing one event}\}.$$

For a global state S, $pred(S)$ and $succ(S)$ can be computed by using *vector clocks* [4], [12]. Every process maintains a vector clock consisting of n components. Let V_i be the vector clock of process P_i. Process P_i increments the ith component of its vector clock, $V_i[i]$, whenever P_i executes an event. When a process sends a message, it timestamps the message with the current value of its vector clock. When P_i receives a message with timestamp T, $V_i[j] = max(V_i[j], T[j])$ for all j. The timestamp of an event e is denoted by $TS(e)$. For an event e in P_i, $TS(e)$ is the value of the updated vector clock V_i when e is executed. Events e in P_i and e' in P_j are *consistent* if $TS(e)[j] \leq TS(e')[j]$ and $TS(e')[i] \leq TS(e)[i]$. When two events are consistent, they can belong to the frontier of a consistent cut and the states after the two events are *concurrent*.

The algorithm for Possibly(Φ) is executed by a monitor process. Each process, after executing an event, sends the timestamp of the event and the values of the local variables related to Φ to the monitor. To see if Φ holds, the monitor visits all the global states (either sequentially or in parallel) and evaluates Φ using the values of the variables stored in the visited global states. From now on, the phrases "visiting a global state" and "testing a global state" have the same meaning as the phrase "testing if Φ holds in that global state."

3 Poly-Space Algorithm for Possibly(Φ)

To detect Possibly(Φ), CM algorithm constructs the lattice of the execution, level by level. The number of global states in the lattice can be exponential in the number of processes. The CM algorithm for Possibly(Φ) traverses the lattice in a breadth first fashion. It generates all the global states in one level, tests them, and then proceeds to the next level. The depth of the lattice is the total number of events in the execution. Thus, the average breadth of the lattice can be

```
    Algorithm DFT(S)
begin
(1)     i = 1;
(2)     Let ⟨k_1, ..., k_n⟩ = S
(3)     while (i ≤ n) do
          begin
(4)         S' = ⟨k_1, ..., k_i + 1, ..., k_n⟩;  (*S', if consistent, is a successor of S *)
(5)         if S' is a global state then
(6)               if value(S) = max(value(pred(S'))) then
(7)                     DFT(S');
(8)         i = i + 1;
          end;
end.
```

Fig. 2. Algorithm for traversing a lattice in a depth first fashion.

exponential in the number of processes. Hence, the space required to store all global states that are in one level in the worst case can be exponential in n.

Our algorithm traverses the lattice in a depth first fashion. To reduce the space complexity, the algorithm does not explicitly store the global states that are already visited. Also, we ensure that each global state is visited exactly once. We perform some computations to decide whether a global state has already been visited or not.

Main Idea. Assume that we are in global state S in the lattice. Let S' be a successor of S in the lattice. Now, we have to decide whether S' is to be visited from S. Global state S' has at most n predecessors in the lattice. All predecessors of S' can be ordered according to their values. The *key rule* for testing a global state exactly once is to visit the global state only from its predecessor that has the maximum value among all of its predecessors.

A formal description of the algorithm for the depth first traversal of a lattice appears in Fig. 2.

Algorithm DFT is a recursive algorithm and is invoked with a global state S. Invoking the algorithm with a global state S means that global state S is visited. For each successor S' of S, the predecessor of S' with maximum value is computed at line six. To find the predecessor of S' with maximum value, all the predecessors of S' are computed by decrementing each component of S' and checking whether the resulting state is a global state. Now, if S is the predecessor of S' with the maximum value, algorithm DFT(S') is recursively invoked. Thus, the algorithm traverses the lattice in a depth first fashion. Algorithm DFT is initially invoked with the initial global state $\langle 0, \ldots, 0 \rangle$.

Example. We illustrate the working of the algorithm with a sample lattice shown in Fig. 1b. The numbers in bold show the order in which the lattice is traversed. The algorithm is invoked with the initial global state $\langle 0, 0 \rangle$ as its parameter. Initially, the value of i is 1. At line 4, $S' = \langle 1, 0 \rangle$. Since $\langle 1, 0 \rangle$ is a global state and $max(value(pred(\langle 1, 0 \rangle)))$ is $\langle 0, 0 \rangle$, DFT($\langle 1, 0 \rangle$) is recursively invoked (i.e., $\langle 1, 0 \rangle$ is visited from $\langle 0, 0 \rangle$).

Now, assume that algorithm DFT($\langle 0, 1 \rangle$) is (recursively) invoked at some intermediate step. At line 4 with $i = 1$, $S' = \langle 1, 1 \rangle$. The set $pred(\langle 1, 1 \rangle)$ is $\{\langle 1, 0 \rangle, \langle 0, 1 \rangle\}$, and $max(value(pred(\langle 1, 1 \rangle)))$ is $\langle 1, 0 \rangle$ and not $\langle 0, 1 \rangle$ (line 6). So, DFT($\langle 1, 1 \rangle$) is not invoked from $\langle 0, 1 \rangle$. When line 4 is executed for the second time, S' becomes $\langle 0, 2 \rangle$. But $\langle 0, 2 \rangle$ is not a global state and since there are no more successors of $\langle 0, 1 \rangle$, the current invocation of the algorithm returns.

Theorem 1. *Assume that a component of the vector timestamp occupies unit space. Then the space complexity of Algorithm DFT is $O(m\ n)$ where m is the total number of events in all of the processes and n is the number of processes.*

Proof Observe that the depth of the recursion for the algorithm is at most m, the depth of the lattice. The size of data stored at each level of recursion is $O(n)$. (The number of components of the timestamp of an event is n). Thus, the space complexity is $O(nm)$. □

A drawback of algorithm DFT is that in an ever-growing lattice the algorithm will not backtrack to check some of the global states at the top of the lattice. Hence, algorithm DFT cannot be used to traverse the lattice of a nonterminating computation.

One way to counter the above mentioned problem is to divide the nonterminating execution into several partial (finite) executions and traverse each partial execution one after another. The lattice for a partial execution has a final global state (or a cut), and a depth first approach can be used to traverse the lattice. After traversing this lattice, consider the execution till another cut that happens at a later time, and traverse the corresponding lattice.

Example. Consider the example in Fig. 3, in which a sample execution and its corresponding lattice are shown. Consider the partial execution E_{C_1}, which ends at cut C_1. The lattice corresponding to E_{C_1} is L_{C_1}. Lattice L_{C_1} begins at global state $\langle 0, 0 \rangle$ and ends at global state $\langle 2, 2 \rangle$. A depth first traversal on lattice L_{C_1} will terminate. Next, the partial execution E_{C_2} which ends at cut C_2 is considered. Let L_{C_2} be the lattice of E_{C_2}. The lattice L_{C_2} ends at global state $\langle 4, 3 \rangle$. Again a depth first traversal on L_{C_2} will terminate. While we traverse the global states in lattice L_{C_2}, it is important not to revisit global states in lattice L_{C_1}. Hence, the depth first traversal of lattice L_{C_2} should not begin from the initial state of lattice L_{C_2} (which is the initial state of L_{C_1} also), it should begin from the end of L_{C_1}. After traversing lattice L_{C_2}, we can proceed to the next partial execution.

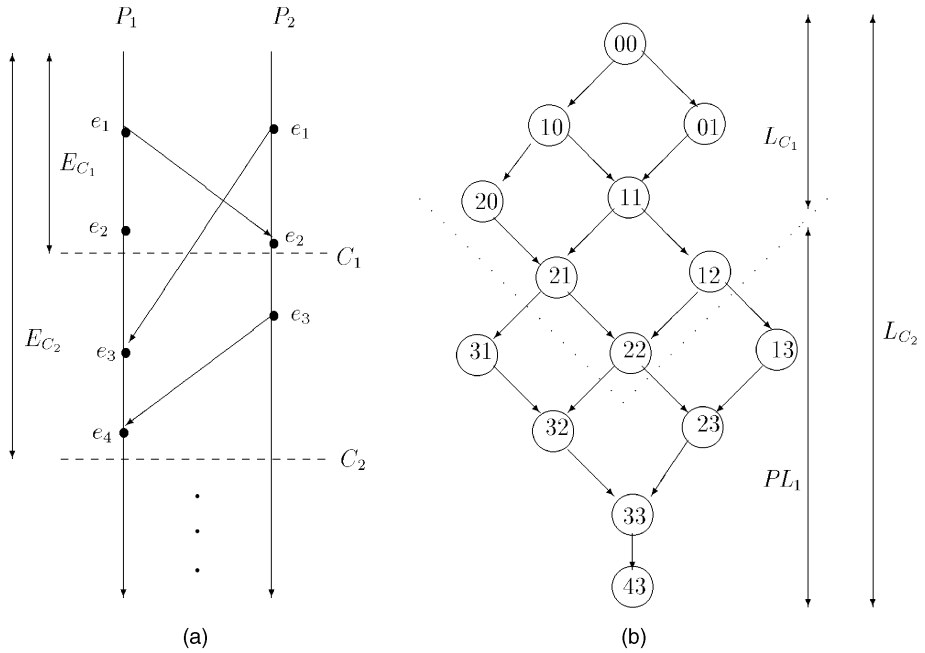

Fig. 3. (a) A sample execution. (b) The lattice for the sample execution.

Let $C_1, C_2, C_3, \ldots$ be the cuts of a nonterminating execution E such that $C_1 \subset C_2 \subset C_3 \ldots$. Let E_{C_i} be the partial execution of the program till cut C_i, and let L_{C_i} denote the lattice corresponding to E_{C_i}. Assume that lattice L_{C_i} has been traversed and lattice $L_{C_{i+1}}$ has to be traversed without revisiting global states that are in L_{C_i}. Let PL_i, which is $L_{C_{i+1}} - L_{C_i}$, be the part of the lattice $L_{C_{i+1}}$ that does not include any global state in L_{C_i}. To traverse the *partial lattice* PL_i, we need to find the starting points of PL_i from which the depth first traversal should begin. The initial global states of a partial lattice are the global states in the partial lattice that do not have a predecessor global state in PL_i. Clearly, the initial global states of a partial lattice are the starting points for traversing the partial lattice. (Note that there can be more than one initial state for a partial lattice. In Fig. 3b, the initial states of partial lattice PL_1 are $\langle 3,1\rangle$ and $\langle 1,3\rangle$.) A straightforward approach to find the initial states of PL_i is to remember the "end states" of lattice L_{C_i}. For example, the end states of L_{C_1} in Fig. 3b is $\{\langle 2,0\rangle$, $\langle 2,1\rangle$, $\langle 2,2\rangle$, $\langle 1,2\rangle\}$. This approach may require exponential space in the worst case. To find these initial states without using excessive storage, let us characterize them further.

For an event e, $mincut(e)$ is the consistent cut C such that $e \in C$ and for any consistent cut C' if $e \in C'$, then $C \subseteq C'$. For any event e, $minstate(e)$ is the global state whose corresponding cut is $mincut(e)$. Clearly, $minstate(e)$ is the "earliest global state" that includes event e. Next, we show that any state that includes event e is reachable from $minstate(e)$ in the lattice. For example, consider event e_3 in process P_1 in Fig. 3. The global state $\langle 3,3\rangle$, which includes event e_3 in P_1, is reachable from $minstate(e_3)$ which is $\langle 3,1\rangle$.

Lemma 2. *Let S be a global state that includes event e and $S \neq minstate(e)$. Then S is reachable from $minstate(e)$ in the lattice.*

Proof. Let C be the cut corresponding to the given global state S which includes event e. Let the cut corresponding to $minstate(e)$ be $mincut(e)$. By the definition of $mincut(e)$ and because $S \neq minstate(e)$, we have $mincut(e) \subseteq C$, which implies that S is reachable from $minstate(e)$. □

Let IS be an initial state of PL_i. Global state IS does not have an immediate predecessor in PL_i but it has an immediate predecessor in L_{C_i}. So, IS is reachable from a global state in L_{C_i} by executing one event. Let e_j in process P_j be such an event. Event e_j is the first event of process P_j that is in execution $E_{C_{i+1}}$ but not in E_{C_i}. Now, we show that $IS = minstate(e_j)$.

Lemma 3. *Let IS be an initial state of the partial lattice PL_i. Let IS include event e_j, which is the first event of process P_j that is in execution $E_{C_{i+1}}$ but not in E_{C_i}. Then, initial global state IS of the partial lattice PL_i is $minstate(e_j)$.*

Proof. Assume to the contrary that IS is not $minstate(e_j)$. Since global state IS includes event e_j, by Lemma 2, global state IS is reachable from $minstate(e_j)$. Observe that $minstate(e_j)$ is a state in partial lattice PL_i. (e_j is an event in execution $E_{C_{i+1}}$ and not in execution E_{C_i}. Therefore, $minstate(e_j)$ is in $L_{C_{i+1}}$ and not L_{C_i}.) Also, all the states on a path from $minstate(e_j)$ to the global state IS are in PL_i since they also include event e_j. Therefore, global state IS has a predecessor state in PL_i. Hence, global state IS cannot be an initial state of PL_i, a contradiction. □

Lemma 3 leads to an efficient way to compute initial states of the partial lattice PL_i. In each process, consider the first event in $E_{C_{i+1}}$ that is not in E_{C_i} The union of the $minstate$ of these events are the initial states of the partial lattice PL_i. This scheme is efficient because it considers

```
    Algorithm INITIAL_STATES(C_{i+1}, C_i)
begin
(1)     Let ⟨e_{k_1}, ..., e_{k_n}⟩ = C_i
(2)     MIN_STATES = ⋃_{j=1}^{n} minstate(e_{k_j+1}) such that e_{k_j+1} ∈ C_{i+1};
(3)     INIT_STATES = { S | S ∈ MIN_STATES and there is no other
                          state in MIN_STATES from which S is reachable}
(4)     return(INIT_STATES);
end.
```

Fig. 4. Algorithm for computing initial states of a partial lattice.

only n events; the straightforward approach considers all the end states of the previous partial lattice. A formal algorithm to compute the initial states of a partial lattice is described in Fig. 4. Variable MIN_STATES contains the earliest global state ($minstate$) of the first event of each process in execution $E_{C_{i+1}} - E_{C_i}$. All the states in MIN_STATES need not necessarily be the initial states of PL_i as some of them may have a predecessor state which belongs to MIN_STATES. Thus, all the states in MIN_STATES that do not have a predecessor state in set MIN_STATES are the initial states of partial lattice PL_i. The maximum cardinality of MIN_STATES is n.

Once we compute the initial states of partial lattice PL_i, we can traverse the entire partial lattice by invoking algorithm PARTIAL_DFT with each initial global state of PL_i. Algorithm PARTIAL_DFT, which is somewhat similar to algorithm DFT in Fig. 2, is described in Fig. 5. Initially, algorithm PARTIAL_DFT tests whether Φ is true at S. If Φ is true at S, the algorithm returns true; otherwise, the algorithm proceeds further. Line 6 of the algorithm ensures that no global state beyond lattice $L_{C_{i+1}}$ is visited. In line 9, only states in $pred(S')$ that are not in L_{C_i} are considered in computing the predecessor of S' with maximum value. This is because S' will not be visited from a global state in L_{C_i} as S' is in lattice $L_{C_{i+1}}$. In the example shown in Fig. 3, global state $\langle 2,3\rangle$ will not be visited from $\langle 2,2\rangle$ since global state $\langle 2,2\rangle$ is in lattice L_{C_1}. Global state $\langle 2,3\rangle$ will be visited from global state $\langle 1,3\rangle$.

Once we know how to detect Possibly(Φ) in a partial lattice, we can handle an ever growing lattice (nonterminating computation) by partitioning the lattice into several partial lattices. A formal algorithm to detect Possibly(Φ) in a nonterminating computation is described in Fig. 6. We assume that cuts $C_1, C_2, \ldots$ that define the partial executions $E_1, E_2, \ldots$ of nonterminating execution E are known. (They can also be provided to the monitor, or the monitor can be made to pick these cuts arbitrarily. A simple way to make the monitor choose is to make the monitor consider x events per process per partial lattice.) The initial states of partial lattice PL_i are computed by invoking algorithm INITIAL_STATES. Then for each initial global state, algorithm PARTIAL_DFT is invoked to test all the global states in PL_i. There is a trade-off in choosing the size of the partial lattice. If the partial lattice is large, then it will take a long time for the algorithm to backtrack and test the earlier global states. However, if the partial lattice is small, the algorithms INITIAL_STATES and PARTIAL_DFT will be invoked often.

```
    Algorithm PARTIAL_DFT(S, C_{i+1}, C_i)
begin
(1)     if Φ is TRUE at S then
(2)         return(TRUE);
(3)     j = 1;
(4)     Let ⟨k_1, ..., k_n⟩ = S
(5)     while (j ≤ n) do
          begin
(6)         if (e_{k_j+1} ∈ C_{i+1}) then
              begin
(7)             S' = ⟨k_1, ..., k_j + 1, ..., k_n⟩
(8)             if S' is a global state then
(9)                 PRED = states in pred(S') that are not in L_{C_i};
(10)                if value(S) = max(value(PRED)) then
(11)                    if (PARTIAL_DFT(S', C_{i+1}, C_i) = TRUE) then
(12)                        return(TRUE);
              end
(13)          j = j + 1;
          end;
(14)    return(false);
end.
```

Fig. 5. Algorithm for detecting Possibly(Φ) in PL_i.

```
    Algorithm POSSIBLY
begin
(1)     if Φ is TRUE at the initial global state ⟨0, ..., 0⟩ then
(2)         return(TRUE);
(3)     i = 0; C_0 = the cut corresponding to global state ⟨0, ..., 0⟩;
(4)     forever do
          begin
(5)         Let C_{i+1} be the cut that defines the end of partial execution E_{C_{i+1}}.
(6)         INIT_STATES = INITIAL_STATES(C_{i+1}, C_i);
(7)         for each S in INIT_STATES do
(8)             if (PARTIAL_DFT(S, C_{i+1}, C_i) = TRUE)
(9)                 return(TRUE);
(10)        i = i + 1;
          end;
end.
```

Fig. 6. Algorithm for detecting Possibly(Φ) in nonterminating computation.

3.1 Correctness and Analysis

Theorem 4. *Possibly(Φ) is true if and only if algorithm* POSSIBLY *in Fig. 6 returns true.*

Proof (⇒). It is sufficient to prove that all global states of the lattice are tested by the algorithm. The proof is by induction on the level of the lattice. The algorithm first tests the initial global state of the computation. Assume that all the global states at level k are tested, for some $k \geq 0$. Consider a global state S' at level $k+1$. Let global state S' be in lattice $L_{C_{i+1}}$ but not in lattice L_{C_i} for some i. Global state S' must have predecessors at level k. But these predecessors may or may not be in lattice L_{C_i}. There are two cases to consider. First, assume that all the predecessors of S' are in lattice L_{C_i}. In this case, S' will be an initial state of PL_i and it will be tested when algorithm PARTIAL_DFT is invoked with S'. Now, consider the case in which some of the predecessors of global state S' are in lattice $L_{C_{i+1}}$. Let S be the global state with maximum value among the predecessors of S' that are in lattice $L_{C_{i+1}}$. Steps 8 through 11 of algorithm PARTIAL_DFT (in Fig. 5) ensures that global state S' will be tested whenever S is tested. By induction hypothesis global state S will be tested since it is at level k. Hence, if Possibly(Φ) is true, algorithm POSSIBLY returns true.

(⇐) Algorithm POSSIBLY returns true only if Φ is true at a global state. Hence, Possibly(Φ) is true. □

Theorem 5. *The space complexity of algorithm* POSSIBLY *is* $O(mn)$ *where* m *is the total number of events in the given execution and* n *is the number of processes assuming* $m > n$.

Proof. The space used by algorithm INITIAL_STATES is $O(n^2)$ (to store the variables MIN_STATES and INIT_STATES). Algorithm POSSIBLY uses $O(mn)$ space (from Theorem 1). Thus, the space complexity of algorithm POSSIBLY is $O(mn)$ assuming $m > n$. □

From Theorem 5, it is clear that a depth first approach offers significant savings in storage. Using the breadth first approach, Possibly(Φ) can be detected early if Φ is true at a global state at the top of the lattice. In a depth first approach, there will be a time delay in detecting Φ at some of the global states that occur early in the execution. However, our algorithm does not proceed purely in a depth first manner. It tests all the global states in a partial lattice before testing any of the global states in the subsequent partial lattice. In that sense, our algorithm also proceeds in a breadth first fashion. By controlling the depth of a partial lattice, our algorithm can be made to test early (but not as early as CM algorithm) the global states that occur early in the execution.

4 Increasing the Granularity of the Execution Step

The performance of algorithms for detecting arbitrary global predicates can be improved by considering a sequence of consecutive events instead of a single event. The value of local variables related to Φ might not change during every event. A consecutive sequence of states in a process in which the values of the local variables related to Φ remain unchanged can be considered together in detecting Φ. An *interval* is a maximal sequence of events such that the values of the local variables related to Φ are the same after the occurrence of every event in the sequence. A process *begins a new interval* if an event changes the value of any local variables related to Φ.

Process P_i maintains an *interval clock* V_i consisting of n components. Process P_i increments the ith component of V_i whenever it begins a new interval. When a process sends a message, it timestamps the message with the current value of its interval clock. When P_i receives a message with timestamp T, $V_i[j]$ is set to $max(V_i[j], T[j])$ for all j. The timestamp of interval I_i is denoted by $TS(I_i)$. For an interval I_i in P_i, $TS(I_i)$ is the value of the updated interval clock V_i when the first event of the interval I_i occurred.

We say that two intervals I_i and I_j of processes i and j, respectively, are *consistent* if $TS(I_i)[j] \leq TS(I_j)[j]$ and $TS(I_j)[i] \leq TS(I_i)[i]$. A *global interval* is a collection of intervals with one interval from every process. A global interval $GI = \langle I_1, \ldots, I_n \rangle$ is consistent if I_i and I_j are consistent for all i, j. The set of all global intervals forms a lattice. A node in a lattice is a global interval, and there is an edge between global intervals GI_i to GI_j if the computation can proceed from one global interval to

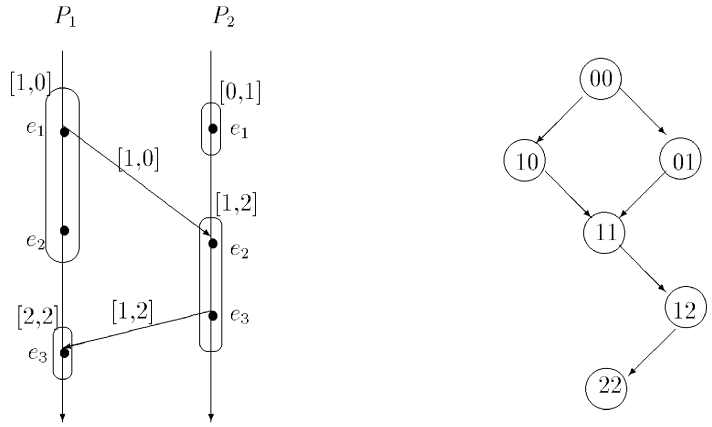

Fig. 7. A sample execution based on interval clocks and its corresponding lattice.

another by executing a sequence of events (interval) in a process.

A sample execution based on interval clocks and its corresponding lattice (based on global intervals) are shown in Fig. 7. (The sample execution is the same as in Fig. 1 except for the addition of intervals and interval clocks.) In process P_1, events e_1 and e_2 belong to one interval and event e_3 belongs to another interval. In process P_2, event e_1 belongs to one interval and events e_2 and e_3 belong to the next interval. The timestamp of each interval is shown at the beginning of each interval. The timestamp of the messages are shown along the messages. The lattice of the sample execution is based on the global intervals generated by the sample execution. By comparing this lattice with the lattice shown in Fig. 1, it is clear that interval clocks can reduce the size of the lattice.

Predicate Φ is true at a global interval if Φ evaluates to true using the value of the variables related to Φ at the global interval. (Note that a variable has a unique value for a given global interval.) We claim that it is sufficient if we test all the global intervals instead of all the global states to detect Possibly(Φ) for an arbitrary predicate.

Let $I.first$ and $I.last$ denote the first event and the last event of the interval I, respectively.

Theorem 6. *There exists a global interval at which Φ is true if and only if there exists a consistent cut (global state) at which Φ is true.*

Proof (⇒). Let $GI = \langle I_1, \ldots, I_n \rangle$ be a global interval at which Φ is true. From the definition of consistent global interval, it is clear that I_i and I_j are consistent, for any i and j. Consider any event e in P_i such that $I_i.last \rightarrow e$. Then, $e \not\rightarrow (I_j.first)$, since I_i and I_j are consistent. Therefore, $e \notin mincut(I_j.first)$. Thus, for any e in P_i

$$I_i.last \rightarrow e \Rightarrow e \notin mincut(I_j.first). \quad (1)$$

Let $(C = \bigcup_{i=1}^{n} mincut(I_i.first))$. C is the supremum of n consistent cuts, hence, C is also a consistent cut [12]. Let the frontier of C be $\langle e_1, \ldots, e_n \rangle$. To prove that Φ is true at C, it is sufficient if we show that e_i is in I_i for all i. Now for any i, either $e_i = I_i.first$, or $I_i.first \rightarrow e_i$. If $e_i = I_i.first$, then e_i is in I_i. Now, consider the case in which $I_i.first \rightarrow e_i$. Since $e_i \in C$, there exists a j such that $e_i \in mincut(I_j.first)$. From (1) (by taking contrapositive) it is clear that $I_i.last \not\rightarrow e_i$. This implies that $e_i \rightarrow I_i.last$, or $e_i = I_i.last$. Therefore, e_i is in I_i. Hence, Φ is true at the consistent cut C.

(⇐) Let $\langle e_1, \ldots, e_n \rangle$ be the frontier of the cut at which Φ is true. Let e_i be in an interval I_i for all i. Consider any i, j. Since e_i and e_j are consistent, for any e in P_j such that $e_j \rightarrow e$, $e \not\rightarrow e_i$. Observe that, $I_i.first = e_i$, or $I_i.first \rightarrow e_i$ and $e_j = I_j.first$, or $e_j \rightarrow I_i.first$. Therefore, $e \not\rightarrow e_i$ (where e is an event such that $I_j.first \rightarrow e$ and e is in P_j). Therefore, $e \not\rightarrow I_i.first$. Hence, $TS(I_i)[j] \leq TS(I_j)[j]$. Similarly, it can be shown that $TS(I_j)[i] \leq TS(I_i)[i]$. Thus, $\langle I_1, \ldots, I_n \rangle$ is a consistent global interval and Φ is true at this global interval. □

Theorem 6 implies that it is sufficient to consider global intervals instead of global states to detect Possibly(Φ). When a process begins a new interval, it sends the timestamp of the interval and the value of the local variable related to Φ to the monitor process. Algorithm POSSIBLY described in Fig. 2 can use global intervals to detect Possibly(Φ). The *successor* and *predecessor* functions can be computed using the timestamps of the intervals. In a process, the number of intervals can be considerably less than the number of events if every event does not change the value of the local variable. Therefore, the total number of global intervals can be substantially less than the total number of global states, improving the performance of our algorithms to detect Possibly(Φ) both in space and time. The linear space algorithm presented in Section 5 cannot use global intervals without increasing the space complexity for the following reasons. The number of intervals that directly depends on an interval can be n in the worst case and, hence, the size of $dep(e)$ can be n. Also, to find whether a global interval is consistent we need an interval clock (whose size is n integers).

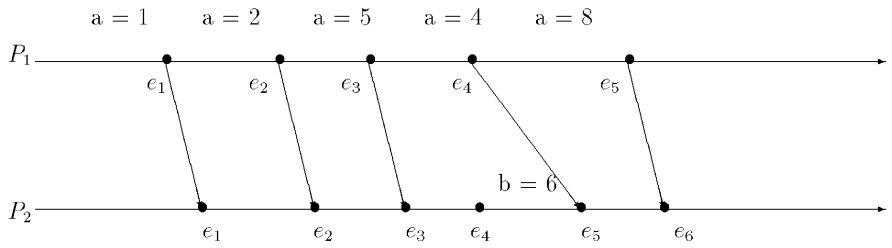

Fig. 8. An example to compare interval clocks and weak vector clocks.

Our concept of interval clock is an extension of weak vector clock used by Marzullo and Neiger [11]. A process P_i increments its ith component (terminates an interval) either when it executes an event that potentially changes Φ, or when it executes a receive event through which it perceives that another process has potentially changed Φ [11]. In our technique, a process terminates an interval only when it executes an event that changes a value of a variable related to Φ. Hence, the number of global intervals in our case can be considerably less than the number of global states obtained by using weak vector clocks.

To compare interval clocks and weak vector clocks, consider the example shown in Fig. 8. The local variable of P_1 is a, and its value is changed by five events. The local variable of P_2 is b, and its value is changed once. (Values a and b are shown only when there is a change.) If we use weak vector clocks, the number of global states that must be tested is 26; whereas the number of global intervals is 9 if we use interval clocks. The difference can be substantial if the number of processes is large.

5 A Linear Space Algorithm

Until now, all the algorithms used vector clocks. We have used vector clocks primarily to check if a given global state is consistent. A close observation of our algorithms reveals that we are not considering global states randomly. At any point of time, from a consistent state, we execute the next event e in each process and check whether the resulting collection of states is a (consistent) global state. The current state and the new global state differ by one event, say, e. The new state is consistent if it includes all the events on which event e depends. So, if we can check whether the direct dependents of e are included in the current global state, then vector clocks are not needed. Using this idea, we eliminate the need for vector clocks and reduce the space complexity further. The linear space algorithm is useful in systems where vector clocks are unavailable. For every event e, the linear space algorithm needs those events that immediately "happened before" e. Let the set of events that immediately happened before e be denoted by $dep(e)$. If e is an internal event or send event, $dep(e)$ contains only the event that occurred immediately before e in the same process. If e is a receive event, $dep(e)$ contains two events—the corresponding send event and the event that occurred immediately before e in the same process. During the computation, we append, to every message, the event number[1] of the corresponding send event and the process in which the send event occurred. Thus, $dep(e)$ can be easily computed on the fly.

Event $e \in C$ is said to be *maximal* if $e \not\rightarrow e'$ for every event $e' \in C$ ($C \subseteq E$). Let $maximal(S)$ denote the set of maximal events in the cut corresponding to the global state S. The set $maximal(S)$ can have at most one event per process. We order the events in $maximal(S)$ according to the id of the process in which they occur. Let $max(maximal(S))$ denote the event in $maximal(S)$ that occurred in the process with the largest id among all the events in $maximal(S)$.

Lemma 7. *Let S be a global state in execution E. Global state S' is a predecessor of S in the lattice of E if and only if S is reachable from S' by executing an event $e \in maximal(S)$.*

Proof ($\Rightarrow$). Let C' and C be the cuts corresponding to S' and S, respectively. Since S' is a predecessor of S, global state S is reachable from S' by executing an event e. Assume that $e \notin maximal(S)$. This implies that there exists an event $e' \in C$ such that $e \rightarrow e'$. Since $C' \subset C$ and $C - C' = \{e\}$, $e' \in C'$. Now, $e' \in C'$ and $e \notin C'$ implies that C' is not a consistent cut and S' is not a global state, a contradiction. ($\Leftarrow$): Follows from the definition of predecessor. □

The above lemma and the ordering of events in $maximal(S)$ suggest a way to test the global state S exactly once. If we are in S', we will test S only if we execute event e at S' to reach S where $e = max(maximal(S))$. For this, the knowledge of $maximal(S)$ is needed. Assuming that we know $maximal(S')$, the following lemma shows a way to compute $maximal(S)$.

Lemma 8. *Let global state S be reachable from global state S' in the lattice of E by executing event e. Then $maximal(S) = maximal(S') - dep(e) + \{e\}$.*

Proof. First, we show by contradiction that for any event e',

$$e' \in maximal(S) \Rightarrow e' \in maximal(S') - dep(e) + \{e\}.$$

Assume that $e' \in maximal(S)$ but

$$e' \notin maximal(S') - dep(e) + \{e\}.$$

Clearly, $e' \neq e$. Also, $e' \not\rightarrow e$ since e' ($\in maximal(S)$) is a maximal event in C and $e \in C$. Thus, $e' \notin dep(e)$. Therefore, $e' \notin maximal(S')$. As e' is an not a maximal event in

1. The *event number* of an event in a process is i if it is the ith event in the process.

```
Global variables:
GS : array [1,...,n] of integers; {contains the current global state}
INC : set of events; {contains the maximal events in GS}
ES : array [1,...,n] of integers; {Contains last state of the current partial lattice PL_i}
Algorithm LINEAR_DFT();
begin
(1)     if Φ is true at GS then
(2)         return(true);
(3)     j = 1;
(4)     while (j < n) do
          begin
(5)         GS[j] = GS[j] + 1; Let e be the GS[j]^th event executed in P_j;
(6)         if GS[j] ≤ ES[j] and GS is a global state then
(7)             temp = INC ∩ dep(e);
(8)             INC = INC − temp + {e};
(9)             if max(INC) = e then
(10)                if (LINEAR_DFT() = true) then
(11)                    return(true);
(12)            INC = INC + temp − {e};
(13)        GS[j] = GS[j] − 1;
(14)        j = j + 1;
          end;
(15)    return(false);
end.
```

Fig. 9. Linear Space Algorithm LINEAR_DFT.

C' there exists an event e'' in C' such that $e' \rightarrow e''$. Since $C' \subset C$ and $C - C' = \{e\}$, $e'' \in C$ also. Therefore, e' is not a maximal event in C. Hence, $e' \notin maximal(S)$, a contradiction.

Next, we show that

$$e' \in maximal(S') - dep(e) + \{e\} \Rightarrow e' \in maximal(S).$$

If $e' = e$, the claim follows from Lemma 7. Now, assume $e' \neq e$. Clearly, $e' \notin dep(e)$. Otherwise,

$$e' \in maximal(S') - dep(e) + \{e\}$$

is not true, since $dep(e)$ is subtracted from $maximal(S')$. So, $e' \in maximal(S')$. Now we claim that $e' \not\rightarrow e$. If $e' \rightarrow e$, then either e' immediately happened before e or $e' \rightarrow e'' \rightarrow e$ where $e'' \in dep(e)$. But $e' \notin dep(e)$. Therefore, $e' \rightarrow e'' \rightarrow e$ where $e'' \in dep(e)$. Now $dep(e) \subset C'$, since $dep(e) \subset C$ and $C = C' \cup \{e\}$. $dep(e) \subset C'$ and $e' \in maximal(S')$ implies that $e' \not\rightarrow e''$. So, $e' \not\rightarrow e$. Therefore, there does not exist any event e'' such that $e'' \in C' \cup \{e\} = C$ and $e' \rightarrow e''$. Hence, $e' \in maximal(S)$. □

We use the above lemma to reduce the space complexity of the algorithm to test a partial lattice. The algorithm, which is recursive, is similar to the algorithm PARTIAL_DFT in Fig. 5. The only difference is in the way the key rule to visit a global state is implemented. A formal presentation of the algorithm is shown in Fig. 9. GS and INC are global variables. GS is the current global state under consideration (to be tested) and is initialized to the initial global state, and INC has the set of maximal events of GS, and is initially empty. To test the entire partial lattice, algorithm LINEAR_DFT has to be invoked $|I|$ times at the top level of recursion where I is the set of initial global states of the partial lattice. For each global state $S \in I$, set GS to S and invoke algorithm LINEAR_DFT. At line 5, we execute the next event, say e, in process i by incrementing the ith component of GS. At line 6, the algorithm checks whether GS is in the current partial lattice and it is consistent. To check that the new value of GS is a global state, it is sufficient to check whether $dep(e)$ is included in GS since the previous value of GS is a global state. In line 8, the maximal events of the current global state (GS) are computed (using Lemma 8). Variable $temp$ is used to store the events removed from INC, so that, INC can be restored to its old value, if necessary. If the maximal event of GS with maximum value is e, algorithm LINEAR_DFT is recursively invoked to continue the depth first search. If LINEAR_DFT() does not return true in line 10, INC and GS are restored to their old values and the algorithm continues with the next successor of GS.

5.1 Correctness and Analysis

Theorem 9. *Possibly(Φ) is true in a partial lattice PL_i if and only if algorithm* LINEAR_DFT *in Fig. 9 returns true when invoked with each initial global state of PL_i.*

Proof ($\Rightarrow$). We prove the contrapositive, i.e., if the algorithm returns false, then Possibly(Φ) is false. To prove this, it is sufficient if we prove that all global states of the lattice are tested by the algorithm. The proof is by induction on the level of the lattice. The algorithm is initially invoked with GS set to the each of the initial global state. So, all global states at level 0 are tested. Assume that all the global states at level k are tested, for some $k \geq 0$. Consider a global state S at level $k+1$. Let $e = max(maximal(S))$ and e be an event in P_i. From Lemma 7, there exists a predecessor, say S', from which S can be reached by executing e. The global state S' is at level k, and by induction hypothesis algorithm LINEAR_DFT is invoked with GS set to S'. Steps 5

through 9 of the algorithm ensures S will also be tested. (At line 5, S will be considered when $j = t$. Since S is a global state, the algorithm will proceed to line 7. From Lemma 8, it follows that the set INC will be equal to $maximal(S)$ after line 8. Since

$$e = max(maximal(S)) = max(INC),$$

algorithm will be recursively invoked with GS set to S, which implies that S will be tested.) Hence, if Possibly(Φ) is true, algorithm LINEAR_DFT returns true. ($\Leftarrow$) Algorithm LINEAR_DFT returns true only if Φ is true at a global state. Hence, Possibly(Φ) is true. □

Theorem 10. *The space complexity of Algorithm* LINEAR_DFT *in Fig. 9 is $O(m)$ where m is the total number of events in the given partial lattice assuming $m > n$.*

Proof. Observe that the algorithm does not use vector clocks to check if a global state is consistent. We store only $dep(e)$, which may contain at most two events, thus using a constant space for every event e. The local variable $temp$ inside the algorithm LINEAR_DFT can contain at most two events. So every time the function is invoked, only a constant amount of additional storage is used. Since the depth of recursion is equal to the total number of events m, the space complexity of the algorithm is $O(m)$. □

Algorithm LINEAR_DFT can be used to test a particular partial lattice. To test the entire lattice of a nonterminating computation, an algorithm similar to the algorithm POSSIBLY in Fig. 6 can be used.

6 Conclusion

6.1 Related Work

Our work was inspired by the works of Spezialetti and Kearns, and Cooper and Marzullo. Spezialetti [13] and Spezialetti and Kearns [14] introduced the concept of *event occurrence*, which was strengthened as Possibly(Φ) by Cooper and Marzullo [3]. Several other researchers—Alagar and Venkatesan [1], Garg and Waldecker [5], Manabe and Imase [10], and Venkatesan and Dathan [18]—present efficient algorithms for detecting Possibly(Φ) by restricting Φ. Stoller and Schneider [15] combine the approach of Garg and Waldecker [5] with any approach that constructs the lattice and present a new algorithm. Their algorithm has the best features of both approaches and improves on each approach. Algorithms for detecting a class of predicates, called atomic sequence of predicates, are presented by Hurfin et al. [6]. These sequences are defined by a pair of sequences of local predicates: Expected predicates and forbidden predicates [6].

Jegou et al. [7] also present a linear space algorithm for online detection of global predicates. The algorithm uses concepts from ordered sets and is based on a particular spanning tree of the ideal lattice called an ideal tree. Our linear space algorithm is simpler and easier to understand than their algorithm.

6.2 Future Directions

In this paper, we have presented several techniques to tackle global state explosion problem in detecting Φ. It will be interesting to explore whether the techniques presented in this paper can be used to detect Definitely(Φ). (Definitely(Φ) is true if, for each computation in the given lattice, Φ is true at a global state of that computation.) We also like to expand Φ to include temporal properties and see whether techniques presented in this paper can be applied for detecting Possibly(Φ). Also, our works have not been implemented. Results of an actual implementation may provide more insight into the global state explosion problem and help in further alleviating the effects of this problem.

Stoller et. al. [17] use partial order methods to reduce the time and space complexities of detecting Possibly(Φ). If two events e_1 and e_2 are concurrent, then, in many cases, it is not necessary to consider both orderings, 1. e_1 followed by e_2 and 2. e_2 followed by e_1. If our DFS algorithm can be modified to incorporate persistent sets [17], then further reduction in time (and space) is possible.

Acknowledgments

The authors would like to thank the three anonymous referees for detailed comments that greatly improved the presentation of this paper. A preliminary version of this paper appears in the *IEEE Proceeding of the International Conference on Parallel and Distributed Systems*, Tiawan, December, 1994. This research was supported in part by the Texas Advanced Technology Program under grant no. 9741-052.

References

[1] S. Alagar and S. Venkatesan, "Hierarchy in Testing Distributed Programs," *Lecture Notes in Computer Science,* vol. 749, pp. 101-116, 1993.

[2] K. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans. Computer Systems,* vol. 3, no. 1, pp. 63-75, 1985.

[3] R. Cooper and K. Marzullo, "Consistent Detection of Global Predicates," *Sigplan Notices,* pp. 167-174, 1991.

[4] J. Fidge, "Timestamps in Message Passing Systems that Preserve the Partial Ordering," *Proc. 11th Australian Computer Science Conf.,* pp. 55-66, 1988.

[5] V. Garg and B. Waldecker, "Detection of Strong Unstable Predicates in Distributed Programs," *IEEE Trans. Parallel and Distributed Systems,* vol. 7, no. 12, pp. 1323-1333, 1996.

[6] M. Hurfin, N. Plouzeau, and M. Raynal, "Detecting Atomic Sequences of Predicates in Distributed Computations," *Proc. ACM/ONR Workshop Parallel and Distributed Debugging,* pp. 32-42, 1993.

[7] R. Jegou, R. Medina, and L. Nourine, "Linear Space Algorithm for On-Line Detection of Global Predicates," *Proc. Int'l Workshop Structures in Concurrency Theory,* 1995.

[8] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM,* vol. 21, no. 7, pp. 558-565, 1978.

[9] H.F. Li and B. Dash, "Detection of Safety Violations in Distributed Systems," *Proc. 1992 Int'l Conf. Parallel and Distributed Systems,* pp. 275-282, 1992.

[10] Y. Manabe and M. Imase, "Global Conditions in Debugging Distributed Programs," *J. Parallel and Distributed Computing,* pp. 62-69, 1992.

[11] K. Marzullo and G. Neiger, "Detection of Global State Predicates,Distributed Algorithms," *Proc. Third Int'l Workshop,* pp. 254-272, 1991.

[12] F. Mattern, "Virtual Time and Global States of Distributed Systems," *Parallel and Distributed Algorithms: Proc. Int'l Workshop Parallel and Distributed Algorithms,* M. Cosnard et. al., eds., pp. 215-226, 1989.
[13] M. Spezialetti, "A Generalized Approach to Monitoring Distributed Computations for Event Occurrences," PhD thesis, Univ. of Pittsburgh, Penn. 1989.
[14] M. Spezialetti and P. Kearns, "Simultaneous Regions: A Framework for the Consistent Monitoring of Distributed Computations" *IEEE Proc. Ninth Int'l Conf. Distributed Computing Systems,* pp. 61-68, 1989.
[15] S.D. Stoller and F.B. Schneider, "Faster Possibility Detection by Combining Two Approaches," *Proc. Ninth Int'l Workshop Distributed Algorithms,* pp. 318-332, 1995.
[16] S.D. Stoller and Y.A. Liu, "Efficient Symbolic Detection of Global Properties in Distributed Systems," *Proc. 10th Int'l Conf. Computer-Aided Verification (CAV '98),* 1998.
[17] S.D. Stoller, L. Unnikrishnan, and Y.L. Liu, "Efficient Detection of Global Properties in Distributed Systems Using Partial-Order Methods," Technical Report 523, Computer Science Dept., Indiana Univ., Oct. 1999,
[18] S. Venkatesan and B. Dathan, "Testing and Debugging Distributed Programs Distributively" *IEEE Trans. Software Eng.,* vol. 21, no. 2, pp. 163-177, Feb. 1995.

Sridhar Alagar received the BSc degree in physics in 1987 from Madurai Kamaraj University and the BE degree in computer science in 1990 from the Indian Institute of Science. He obtained the PhD degree in computer science in 1995 from the University of Texas at Dallas. From May 1990 to July 1991, he worked as a software engineer at PSI Data Systems Ltd., Bangalore. He worked at Alcatel Network Systems from September 1994 to August 1998, as a senior research scientist. Currently, he is working for ONI Systems. His research interests include survivable optical networks, fault-tolerant distributed systems, mobile computing, and testing distributed systems.

Subbarayan Venkatesan received the BTech degree in civil engineering and the MTech degree in computer science from the Indian Institute of Technology, Madras in 1981 and 1983, respectively. He completed the PhD degree in computer science from the University of Pittsburgh in December 1988. In January 1989, he joined the University of Texas at Dallas where he is currently an associate professor of computer science. His research interests are in distributed systems, mobile and wireless networks, and testing and debugging distributed programs. He is a member of the IEEE Computer Society.