Efficient Global Event Predicate Detection

Hsien-Kuang Chiou*, Willard Korfhage

Computer Science Dept., Polytechnic University, Brooklyn, NY 11201

Abstract

Detection of global event predicates is an important issue for distributed systems, particularly for debugging and monitoring of such systems. This paper defines event normal form (ENF) event predicates, and two on-line algorithms to detect the first occurrence of such predicates. These algorithms differ in the technique used to match concurrent groups of events. The first builds and searches a tree of a possible matches. The second algorithm achieves the same result as the first, but needs only a single node of the tree at any time, resulting in better space and time complexity. We outline correctness proofs, and provide performance measurements from an implementation.

Keywords: event normal form, event predicate, distributed algorithm, distributed debugging

1 Introduction

An event is an occurrence of interest in a process. It may be a variable being assigned a certain value, a call to a particular subroutine, sending or receiving a message, or any occurrence that can be specified by an action and/or the state of a process. A global event predicate is a boolean expression over events in different processes in a distributed or parallel system. Event predicates may be stable (once true they remain true, such as a predicate for a deadlock), or unstable (a predicate may be true at one time, and false at another). Detecting global event predicates in distributed systems is an important issue in several areas such as monitoring, debugging, and performance analysis [3][7][10].

The major contributions of this paper are to define event normal form (ENF) for event predicates, to provide two on-line distributed algorithms to detect the first match of an unstable ENF predicate, and to outline correctness proofs for these algorithms.

The idea behind the distributed matching algorithms is to decompose a global event predicate into local predicates which can be detected by local processes in parallel. These local processes communicate with other processes to achieve the global predicate detection. The basic matching algorithm constructs a tree of potential pattern matches, and searches this tree in breadth-first order to find an actual match. In the first algorithm we give, this tree grows exponentially quickly, and without bound. By making use of temporal properties of the nodes of the tree, we create a second algorithm that finds the same match as the original algorithm, but needs only a single node of the tree in memory at any time. In addition the second algorithm does not process any events that temporally follow the match.

In the remainder of this paper, Section 2 discusses related work, Section 3 defines a general event predicate, Section 4 introduces the first pattern matching algorithm, and Section 5 provides the second algorithm. We discuss performance in Section 6.

2 Related work

The work most related to this paper is by Garg and Waldecker [5]. The authors characterize distributed predicates, and detect them by decomposing a distributed predicate into local predicates. Their algorithm is similar to ours, and can detect unstable predicates. Our work differs from theirs in the following ways: 1) Their work can detect sequential predicates or conjunctive predicates, but not combinations of these. 2) Their algorithm is centralized. All events go to a single process. 3) Their algorithm processes every event that arrives. Our algorithm tracks causal dependencies between processes and attempts to examine only the events necessary to match the predicate.

Cooper and Marzullo [3] introduced some algorithms to detect whether global state predicates are true. They considered a possible execution of the system as a total

* Mr. Chiou was partially supported by a UniForum Research Award.

order of global state in which one process takes a step between adjacent global states. However, the number of global states is $O(k^n)$, where k is the maximum number of events in a process, and n is the number of processes. This huge number of states makes the technique computationally expensive.

Haban and Weigel [7] defined and detected global events in distributed systems. Their detection algorithm is based on the comparisons of the timestamps of the events involved in the relation. In their work, the time at which a global predicate is satisfied is the time of the last event in the predicate to match. This can lead to inconsistency in more complex predicates, as described in [8].

Miller and Choi [10] introduced an algorithm to detect "linked predicates" in which the event ordering can be specified. Their algorithm cannot detect concurrent events.

Other approaches, like taking a global snapshots to detect stable predicate, have appeared in papers like [2] [6]. An issue for this approach is how often we should take snapshots so interesting behavior will not be missed. The difficulty detecting unstable predicates limits the application of this approach.

3 Defining general event predicates

3.1 Logical time

Distributed systems do not have a global clock, so we must use logical time to determine event causalities.We use vector clocks, as defined by Fidge [4] and Mattern [9], for this purpose. The following section defines vector clocks as they did.

Timestamping mechanism

We assume that there are n processes participating in the computation. Each process has its own local clock and maintains a vector timestamp consisting of n components, each of which corresponds to one of the participating processes. The value of each component represents last known clock value for the corresponding process. The timestamping rules work as follows:

- All values are zero initially.
- The local clock (C_i) increments by at least one before any event occurs in process P_i.
- When sending a message, append the current timestamp to the message.
- When receiving a message, the receiving process updates its timestamp by taking component-wise maximum between its own vector timestamp and the timestamp appended to incoming message.

In our algorithm, whenever a local event predicate is satisfied, an "event message" is sent to a checking process, and the event message contains a copy of the local vector timestamp at the time the local event predicate was satisfied.

3.2 General event predicates

We now define relationships between events in terms of vector time. Definitions 1 and 2 below have appeared before, whereas definitions 3 and beyond are new. We use the following notation:

- $E = E_1 \cup E_2 \cup ... \cup E_n$ denotes the set of events have occurred at processes P_1 through P_n.
- A primitive event is any occurrence of interest in a local process. We let $e_i \in E_i$ denote any generic primitive event happening in process P_i, and $e_{i,j}$ be the jth event in P_i. For a primitive event $e \in E$ in an arbitrary process, we use the function $P(e)$ to return the index of the process generating e (i.e. event e happened in process $P(e)$).
- $V(e)$ denotes the vector timestamp for event e, and $V(e, i)$ denotes the component value corresponding to process i in $V(e)$.

Definition 1: Sequential relationship "$\rightarrow$."

For two different events $e, f \in E$, $e \rightarrow f$ iff $V(e, P(e)) \leq V(f, P(e))$.

Definition 2: Concurrent relationship "$\wedge$."

For two events $e_i, e_j \in E$, $e_i \wedge e_j$ iff $\neg(e_i \rightarrow e_j)$ and $\neg(e_j \rightarrow e_i)$. Because there is no "transitivity" property for the "$\wedge$" relation, a set of events is concurrent if and only if every pair of events in the set is concurrent.

Definition 3: "$\rightarrow$" between sets of concurrent events.

Let $S_1 = \{e_{i_1}, e_{i_2}, ..., e_{i_k}\}$ be a set of concurrent events, and $S_2 = \{e_{j_1}, e_{j_2}, ..., e_{j_l}\}$ be another set of concurrent events. We define: $S_1 \rightarrow S_2$ iff $\forall e \in S_1$, $V(e, P(e)) \leq \text{Min}(V(f, P(e)), \forall f \in S_2)$. By definition 1, this means that every event in S_1 precedes every event in S_2.

Definition 4: Group Set.

A *group set* $G = \{S_1, S_2, ..., S_m\}$ is a set of sets of concurrent events, such that each S_i in a G is a set of concurrent events and every S_i consists of events from same set of processes $P = \{1,2, ..., k\}$. If there is a com-

ponent set of G which causally precedes every other set in G, then we define this as the *first set* of G, denoted S_F. More formally, let S_x be any set in G other than S_F, if we denote e_b^x as the event in S_x from process b, then $\forall i \in P$, $e_i^F = e_i^x$ or $e_i^F \rightarrow e_i^x$.

3.3 Event normal form predicates

A primitive event predicate is an expression that matches a single event. Primitive event predicates can be combined to form complex event predicates using concurrent, sequential, and alternative relations ("$\wedge$," "$\rightarrow$," and "$|$," respectively). It can be difficult to define the semantic notion of time for an arbitrary, complex event predicate using these relations, and instead we create event normal form (ENF) predicates, which are based on sequential relationships between groups of concurrent predicates. We can transform many arbitrary event predicates into ENF, as described in [8].

Definition 5: Concurrent Event String (CES).

A concurrent event string is a primitive event predicate, or an event predicate with the following form: CES = $ep_1 \wedge ep_2 \wedge \ldots \wedge ep_k$. This predicate is true iff there is a set of concurrent events $e_1, e_2, \ldots, e_k$ such that e_i matches ep_i, $\forall i = 1..k$.

Many sets of events may satisfy a CES. These sets of events form a group set, say G_c. The *first match* of a CES is first set of G_c as given by definition 4.

Definition 6: Sequential Event String (SES).

A sequential event string is a CES or an event predicate with the following form: ConcurrentEventString $\rightarrow$ ConcurrentEventString $\rightarrow$ ... $\rightarrow$ ConcurrentEventString. We say that a sequential event string $CES_i \rightarrow CES_j$ is true iff there are two sets of concurrent events, S_1, matching CES_i, and S_2, matching CES_j, and $S_1 \rightarrow S_2$.

For a general SES = $CES_1 \rightarrow CES_2 \rightarrow \ldots \rightarrow CES_n$, there may be many potential matches for each CES_i, and consequently many possible matches for the SES. We define the *first match* of a SES to be the first match of each CES that also satisfies the required precedence relationships between the CES's.

Definition 7: Event Normal Form.

An ENF predicate is a SES or an event predicate with the following form: $SequentialEventString_1 \mid SequentialEventString_2 \mid \ldots \mid SequentialEventString_k$. This predicate is true iff any $SequentialEventString_i$ in the ENF is true.

4 Detecting ENF predicates

In this section we present an algorithm to detect the first occurrence of a single ENF predicate given in event normal form. This algorithm could be used to trigger a debugging breakpoint, or to automatically initiate a management task, but the systems to perform such tasks are outside the scope of this paper. We make following assumptions:

- Each process can communicate with any other process.
- Message delivery is FIFO.
- There is no global clock, and the processes involved maintain vector timestamps.

We will describe the actions of this matching algorithm and explain how it works. Pseudocode for the algorithm can be found in [8]. The algorithm uses three types of processes:

- A master process, that receives the initial ENF and initiates the algorithm.
- CES matching (CESM) processes that detect concurrent event strings.
- Local event matching (LEM) processes that detect the local event predicates which compose a concurrent event string.

Intuitively, the LEM looks for individual events, and the CESM sifts through these events to find a group of concurrent events that matches its pattern.

For each sequential event string in the ENF predicate, the master process distributes the SES's concurrent event string components to CESM processes it creates. Each CESM process then creates local event matching processes for all the machines or processes involved in the CES. Figure 1 illustrates the creation of processes and distribution of an event pattern of the form $(e_1 \wedge e_2) \rightarrow e_3$. For purposes of presentation, we show one CESM process residing on one of the processors involved in its concurrent event string, but it may run anywhere in the system.

The LEM processes send event messages to their CESM process whenever the local event predicate is satisfied. These event messages contain a copy of the local vector timestamp at the time when the predicate was satisfied.

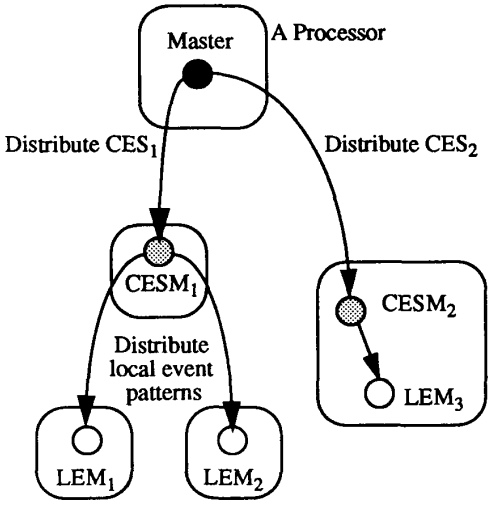

FIGURE 1. Distributing the event pattern.

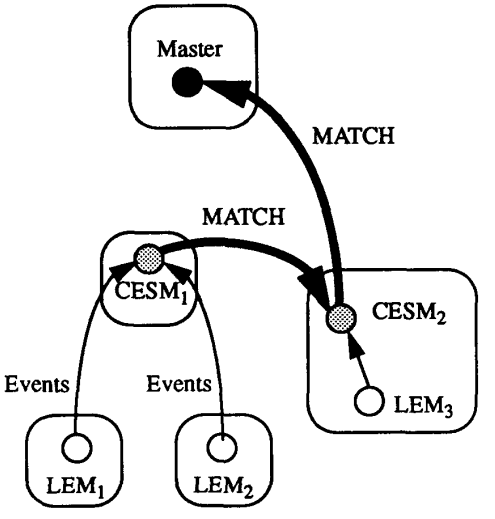

FIGURE 2. Matching the event pattern.

When the first CESM process ($CESM_1$) finds a concurrent set of events from the LEM processes, it sends a MATCH message containing this set of events to the next CESM process ($CESM_2$). This second CESM process also looks for a concurrent set of events from its LEM processes. When it finds a concurrent set that follows (in the temporal sense) the set of events from the preceding CESM process, the second CESM process sends a MATCH message containing its own set of concurrent events to the next CESM process. The last CESM process sends a MATCH message to the master process, at which point the ENF pattern has been matched. Figure 2 illustrates this message flow.

The design of the master process is straightforward, and the LEM processes simply test a boolean predicate against local process or machine state, but the CESM processes are more complex, and we will discuss their matching algorithm in more detail.

4.1 Concurrent event string matching

We match a concurrent event string by constructing a search-tree of event sets. Each node in the tree is a set of events containing one event from each process involved in the concurrent event string. The root of the tree is the first set of concurrent events from the LEM processes, and the other nodes in the tree are built from their parents by changing one event, say e_{ij}, to the next event for that process, $e_{i(j+1)}$.

Each CESM process has a queue (Q_x) corresponding to each LEM process. Event e_x from LEM_x is enqueued in queue Q_x. Concurrent event string matching goes through two phases at each CESM: finding the first concurrent set of events, then constructing the search-tree.

The CESM detects the first set of concurrent events by comparing events at top of each queue pair-wise until the events at top of all queues are mutually concurrent. During this step, because of the FIFO assumption, we can eliminate the earlier event on each comparison if a precedence relationship is found. This part is identical to the algorithm of [5].

Once the CESM finds the first concurrent set of events, it constructs the search-tree. The root node of this tree is the first set of concurrent events. The algorithm builds the search-tree in "breadth-first" order: when a new event comes in, it traverses each level of the tree from top to bottom, adding new nodes whenever possible. The algorithm keeps an event counter, initialized to 0 and increased by one for each new event received (after it has found the first concurrent set). Each node is assigned the current event count when it is created.

As the algorithm constructs the tree, it also checks nodes to see if they contain sets of concurrent events. Starting from the unchecked node with the lowest event count, the algorithm checks all nodes with the same event count in breadth-first order to see if each node is a set of concurrent events. Nodes that do represent concurrent events are placed into a queue called the *concurrentQueue*.

When the CESM receives a MATCH message from the previous CESM, it dequeues nodes from the concurrentQueue one by one, checking whether each is preceded by the set of events in the MATCH message (also called the *previous set*). If the CESM finds a node which follows the previous set, then it sends a MATCH message containing the node to the next CESM process. Until this is true, the

CESM continues to process incoming events, build the search-tree, and test concurrent nodes against the previous set.

We illustrate this algorithm operation with the example shown in Figure 4, using the events (shown with their vector timestamps) shown in Figure 3. The arrows in Figure 3 denote communication events (to indicate the event precedences), but these events are not collected for matching in this example.

Assume this CESM is looking for events from LEM_1 and LEM_2, and the previous set has been received and is the single event e_{31}. We label each search tree node with a letter for convenience in showing its place in the concurrentQueue, and put the event count of each node in parenthesis beside its label.

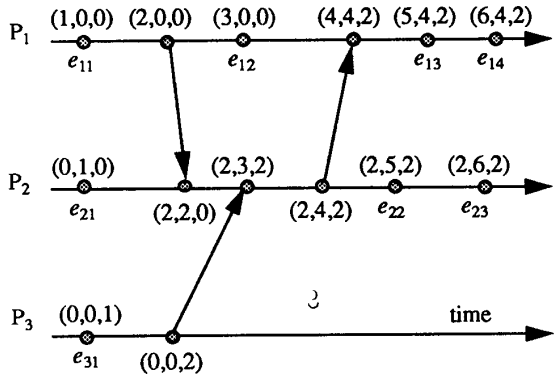

FIGURE 3. Example events

The algorithm builds the tree as it receives and processes events from the queues. Note that the tree contains duplicate nodes (e.g. nodes D and E). As the algorithm builds the tree, it places the concurrent nodes (A, B, D, E, G, H, and I) in concurrentQueue. Because A, B, D, and E are not preceded by the previous set, but G is preceded by the previous set, the algorithm sends a MATCH message when it checks node G.

Proof of correctness:

Because our algorithm enumerates all possible combinations of events from LEM processes, if a match is possible, the algorithm will detect it. Theorem 1 below shows that this match has temporal relationships necessary for the first match. First we define the following notation:

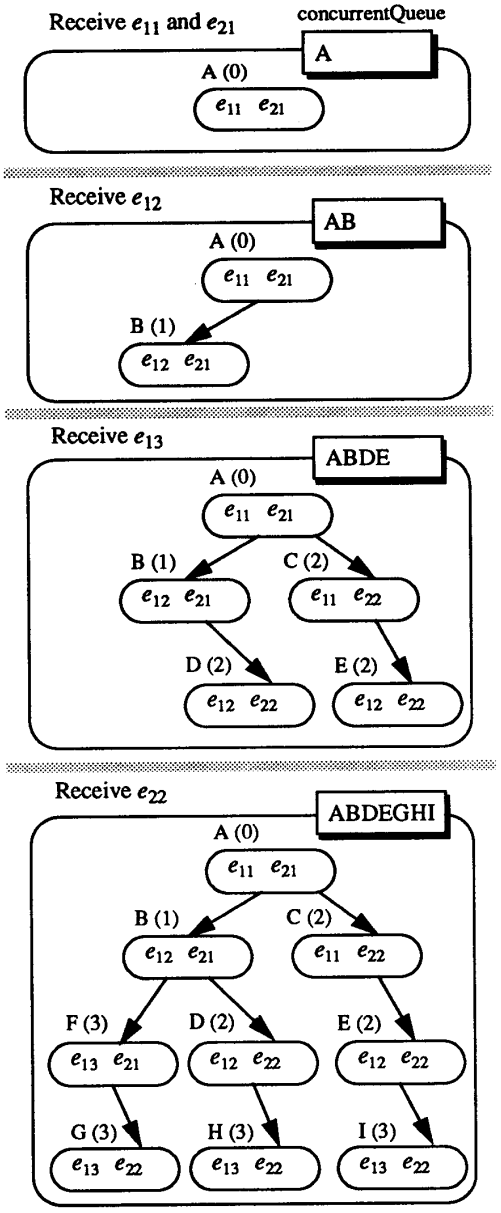

FIGURE 4. Search-tree construction.

- $\eta_w = (e_1^w, e_2^w, ..., e_k^w)$ denotes a node in the search-tree. Event e_i^w is the event in node η_w from process i. We let $P = \{1,2, ..., k\}$ denote the set of processes involved in η_w. We let ξ be the set of all nodes in our search tree.

- $\eta_\alpha = (e_1^\alpha, e_2^\alpha, ..., e_k^\alpha)$ is the node the algorithm picks as the first match of a CES.
- $\eta_{prev} = (e_1^{prev}, e_2^{prev}, ..., e_j^{prev})$ is the previous set.

Theorem 1: After the CESM algorithm finds η_α as the first match, if the algorithm later finds $\eta_\beta \in \xi$, then η_β will meet the following constraints: $\forall i \in P$, $e_i^\alpha = e_i^\beta$ or $e_i^\alpha \rightarrow e_i^\beta$. This theorem shows that η_α is the first match of the CES.

Proof: Because of space constraints, we omit the complete proof by contradiction, which may be found in [8]. In brief, however, we assume that after finding η_α, the algorithm finds η_β such that $\eta_\beta \rightarrow \eta_\alpha$. Using events from η_α and η_β we can construct another node in the search tree that must be detected earlier than η_α, leading to a contradiction.

5 Second algorithm

The previous algorithm has high complexity in both time and space because the search tree has many duplicate nodes, and nodes are never deleted. In particular, the search-tree in the algorithm may have many duplicates of η_α at the same level, and therefore many identical-length paths from the root that reach η_α. Because our goal is satisfied if we find any one of these duplicated nodes, it is sufficient to generate those nodes on a single path to a node η_α. In this section we give two lemmas that tell how this can be done. We use the following functions:

- NextEvent(e_i^x) is the next event matching ep_i after e_i^x.
- $index(e_i^x)$ is the number of events matching ep_i before e_i^x occurred.
- $Index(\eta_x) = \sum_{i \in P} index(e_i^x)$.
- NextNode(η_x, e_i): If $\eta_x = (e_1^x, e_2^x, ..., e_k^x)$, then η_m = NextNode(η_x, e_i) is the node such that $\forall s \in P$, $e_s^m = e_s^x$ for $s \neq i$, and $e_s^m = e_i$ for $s = i$. This function is used to generate the next node of the search tree.

Definition 8: Let $\mu = \{\eta_x = (e_1^x, e_2^x, ..., e_k^x)\}$ denote the set of nodes in our search-tree such that $\forall i \in P$, $e_i^x = e_i^\alpha$ or $e_i^x \rightarrow e_i^\alpha$. Intuitively, μ is the set of nodes in the search tree that are either one of the η_α nodes, or a node which is at a level higher than η_α and has at least one path to an η_α.

Lemma 1: Let e_v = NextEvent(e_v^x) and η_m = NextNode(η_x, e_v). If $\eta_x \in \mu$, and $\neg(\eta_{prev} \rightarrow \{e_v^x\})$, then $\eta_m \in \mu$.

Lemma 2: Let e_r = NextEvent(e_r^x), and η_m = NextNode(η_x, e_r). If $\eta_x \in \mu$, and $\exists r, s \in P$ such that $e_r^x \rightarrow e_s^x$ in η_x, then $\eta_m \in \mu$.

Lemmas 1 and 2 give the conditions for building nodes on a path to η_α. Lemma 1 states that we can build the next node using the next event from any process whose current event does not happen after the previous set. Lemma 2 states that we can build the next node using the next event from any process whose current event precedes some other event in the current node. Thus, by checking events in the current node against the previous set (what we call the *precedence test*), and against other events in the node (the *concurrency test*), we can construct only the nodes that lead to η_α.

Theorem 2: Let $Index(\eta_x)$ = c and $Index(\eta_\alpha)$ = d. If an algorithm that creates only nodes in μ is currently at node η_x, then after d - c steps, the algorithm will match one of those η_α in our original search-tree.

Proof: By Lemma 1 and Lemma 2, each time after we apply a precedence or concurrency test to η_x, we will derive $\eta_m \in \mu$ and $Index(\eta_m) = Index(\eta_x) + 1$. Thus, after c - d steps, we will find a final node η_m^f such that $Index(\eta_m^f) = Index(\eta_\alpha)$. Because $\eta_m^f \in \mu$, and from the properties of $Index(\eta_\alpha)$, we have $\eta_m^f = \eta_\alpha$.

To implement an algorithm using Lemmas 1 and 2, we track which processes' events in the current node fail the precedence test (we say these processes are in NS), and which processes' events in the current node pass the concurrent test (we say these processes are in C). Processes not in NS or in C are in NC. When all processes are in C, the algorithm has found a match. Below we give the second CESM algorithm:

CESM Algorithm
Receive η_{prev}.
Put all processes in NS

while all processes are not in C

Receive an event e_i from a process P_i in NS or NC.
Make the next tree node using this event.

```
If the process was in NS, then
    if η_prev→{e_i} then
        /* e_i passed the precedence test. */
        Create CT, which contains ordered pairs of
            processes involving P_i and all processes
            in C, for use in the concurrency test: if P_j
            is in C, then (P_i, P_j) and (P_j, P_i) are put
            into CT.
        Remove P_i from NS and put it into C.
    else
        Set CT to empty. /* Skip concurrency test. */
    endif
else
    Create CT, which contains the ordered pairs
        involving P_i and all processes in C, for use in
        the concurrency test.
    Remove P_i from NC and put it into C.
endif

/* Concurrency test. */
while CT isn't empty
    Remove an ordered pair, say (P_r,P_s), from CT.

    If the events in the current node for this pair have
    a precedence relationship (e.g. e_r→e_s), then

        Try to receive a new event from process P_r.
        If you can receive a new event, then
            Make the next tree node using the new
              event.
            Put pairs into CT that will test e_r for
                concurrency with events from
                processes in C.
            Put P_r into C.
        else /* No event for process P_r. */
            If P_r is in C then Put P_r into NC.
            Remove all ordered pairs involving P_r
                from CT.
        end if
    end if
    /* Do nothing if no precedence relationship */

    end while /* CT isn't empty */

end while /* not all processes are in C. */

The algorithm found a match. Send the current node to the
    next CESM.
End of CESM algorithm.
```

6 Performance

6.1 Space Complexity

The space complexity of both algorithms depends on space requirements for the search-tree and the queues storing incoming events. As mentioned previously, the search-tree of our first algorithm could expand exponentially. However, in the second algorithm, we need space only for a single node of the tree at any time.

To be more precise, each tree node contains l event messages, where l is the number of LEM's in the CES. Because each event message has a vector time, it requires space for up to p logical clock values, where p is the number of processes in the system. The second algorithm also uses two Boolean arrays, one with l elements and the other with l^2 elements. We expect l to be small under most circumstances, and p will be at least as large as l. Thus the total storage space required for each CESM is $O(lp)$.

For the ENF detection algorithm as a whole, each SES requires only one CESM to be active at a time. If an ENF expression contains s sequential event strings, then the space required for data structures is $O(lp)$ in each of s processes.

The queue space requirements depend on the order of event message arrival. The second algorithm removes one event from some queue every time it tests for precedence or concurrency. However, sometimes the algorithm temporarily receives events only from a subset of the processes involved in the predicate, and must wait for events necessary to generate a new node. At these times, some queues can grow without bound. We discuss some ways to eliminate some of these events when we discuss time complexity.

6.2 Time complexity

Comparing the vector clock of one event to the vector clocks of other events to determine causal relationships ("examining an event") is the most time consuming operation in our algorithm. The fewer events an algorithm examines, the shorter its execution time. Thus, we measure time complexity in terms of the number of events examined.

Our algorithm shows that to match a group of concurrent events, it is sufficient to examine the matching events themselves and all events that precede them (in vector time). In general, however, it is possible to examine fewer events than this by deducing dependencies from events that have already been examined. For example, if $e_{i,j}$ fails the precedence test, then we know that any earlier events from process i will also fail the precedence test. A more complex case is that if $e_{i,j+1}$ depends on $e_{i,j}$ through a chain of causal dependencies involving events in pro-

cesses other than i, then the events in this dependency chain, other than the events at process i, can never be part of a concurrent set of events, and thus need not be examined. However, it may be more time consuming to search for such dependency chains than it is to simply apply our algorithm to the events.

We can show, in certain cases, that the number of events our algorithm examines is close to a lower bound on the problem. If a CES's match is the k^{th} event from each of the l LEM's, then we can prove that any algorithm must examine the vector clocks at least kl - (l - 1) events (versus the kl events our second algorithm examines). We are working on a lower bound for more general cases.

6.3 Performance measurements

We tested the performance of the second CESM algorithm with an implementation on a Sun 4/40 using Concert/C [1], which provides RPC for ANSI C. We sent 10,000 event messages to the CESM from another workstation. These events had a round-robin causal dependency, where the event from LEM i depended on the event from LEM i-1 (modulo the number of LEM's). The dependencies in the final events differ so that they will form a concurrent set. In Figure 5 we illustrate an example with 6 events from 3 LEM's.

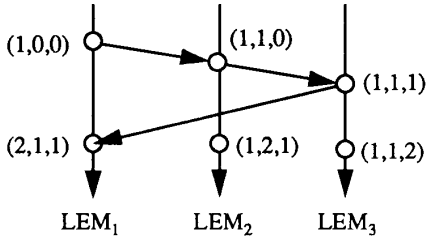

FIGURE 5. Six test messages from 3 LEM's.

The maximum message throughput, without any processing of messages, was 508 messages/second. Table 1 gives the performance with the CESM algorithm, and it is quite good. As expected, performance decreases as the number of LEM's increases because concurrency checks may involve many more comparisons. Nonetheless, even with 8 LEM's, processing an event message once it has been received takes, on average, less than 500 µs.

Table 1: CESM Performance (messages/ second)

Number of LEM's						
2	3	4	5	6	7	8
469	454	447	428	425	416	413

7 Conclusion

In this paper we defined an event normal form (ENF) for event predicates, and presented an efficient distributed algorithm to detect the first match of a global event predicate. We motivated the algorithm's development by initially presenting a simple algorithm that builds a tree of all possible concurrent event sets and searches that tree for a predicate match. By taking advantage of the temporal properties of nodes in the tree, we developed a second algorithm that keeps only a single node of the tree to achieve the same result. The second algorithm is efficient in space and time, and an implementation show very good performance.

References

[1] J. S. Auerbach, A. P. Goldberg, A. S. Gopal, M. T. Kennedy, and J. R. Russel, "Concert/C: A language for distributed programming," USENIX '94.

[2] K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM TOCS*, pp. 63-75, Feb. 1985.

[3] R. Cooper and K. Marzullo. "Consistent Detection of Global Predicates," *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, CA, 1991, pp. 167-174.

[4] C. Fidge, "Timestamps in Message-Passing Systems That Preserve the Partial Ordering," *Proc. 11th Australian Computer Science Conference*, University of Queensland, 1988. pp. 55-66.

[5] V. Garg and B. Waldecker, "Detection of Unstable Predicates in Distributed Programs," Technical Report, University of Texas at Austin, March, 1992.

[6] J.-M. Helary, N. Plouzeau, and M. Raynal, "Computing Particular Snapshots in Distributed Systems," *Proc. of the 9th Annual International Phoenix Conference on Computers and Communications*, Scottsdale, Ariz., 1990. pp. 116-123.

[7] D. Haban and W. Weigel, "Global Events and Global Breakpoints in Distributed System," *Proc. 21th Annual Hawaii Intl. Conference on System Science*, 1988. pp. 166-175.

[8] W. Korfhage and H. Chiou, "Efficient Detection of Global Event Predicates," CATT technical report CATT93-59, 1993.

[9] F. Mattern, "Virtual Time and Global States of Distributed Systems," *Parallel and Distributed Algorithms: Proc. of the International Workshop on Parallel and Distributed Algorithms*, Elsevier Science Publishers, North Holland, 1989, pp. 215-226.

[10] B. Miller, and J.-D. Choi., "Breakpoints and Halting in Distributed Systems," *Proc. 8th Intl. Conference on Distributed Computing Systems*, 1988. pp. 316-323.