An Efficient Distributed Online Algorithm to Detect Strong Conjunctive Predicates

Loon-Been Chen and I-Chen Wu

Abstract—Detecting strong conjunctive predicates is a fundamental problem in debugging and testing distributed programs. A strong conjunctive predicate is a logical statement to represent the desired event of the system. Therefore, if the predicate is not true, an error may occur because the desired event does not happen. Recently, several reported detection algorithms reveal the problem of unbounded state queue growth since the system may generate a huge amount of execution states in a very short time. In order to solve this problem, this paper introduces the notion of removable states which can be disregarded in the sense that detection results still remain correct. A fully distributed algorithm is developed in this paper to perform the detection in an online manner. Based on the notion of removable states, the time complexity of the detection algorithm is improved as the number of states to be evaluated is reduced.

Index Terms—Conjunctive predicate, distributed debugging, distributed system, global predicate detection.

1 INTRODUCTION

With the rapid development of networks and distributed systems, programming in distributed environments has become quite common. However, the difficulty associated with distributed programming is much higher than that of sequential programming. This arises from the fact that distributed debugging requires the capability to analyze and control the execution of processors to be running asynchronously. Also, within a distributed environment, stopping a program at a specific breakpoint is a nontrivial task.

It is well-understood that distributed programs are usually designed to obey certain conditions [1]. For example, a distributed mutual exclusion program obeys the condition "at any time, the number of processes in the critical section is no more than one." If this condition is violated, an error (two or more processes are in the critical section simultaneously) may occur. Typically, the conditions are formulated as Boolean expressions, called *global predicates* [2], [3]. Detecting whether or not a given global predicate is satisfied is essential to debugging and testing the distributed computations.

As the detection of general global predicate was proven to be NP-complete [4], most researchers restricted their research to a specific class of global predicates. In this paper, the focus is on an important class of global predicates, known as *conjunctive predicate* [5], [6], [7], [8], which can be expressed as a conjunctive form of local predicates. This *local predicate* is a Boolean expression defined by the local variables of a process. At any time, this process can evaluate its local predicate without the necessity of communication.

In this paper, the problem of detecting whether a given conjunctive predicate Φ is *definitely true* [3], [8] is considered. Φ is definitely true if, for *all* runs of the distributed program, Φ is true at some time. Intuitively, detecting this sort of global predicates is used to ensure that a certain *desired* event would occur. For simplicity, we define predicate $DEFINITELY(\Phi)$ is true if and only if Φ is definitely true. $DEFINITELY(\Phi)$ is called a *strong* conjunctive predicate in [6].

In [8], Venkatesan and Dathan proposed a distributed algorithm to detect $DEFINITELY(\Phi)$. This algorithm performs an *offline* evaluation of the predicates, i.e., predicates are evaluated after the program execution is terminated. The analysis indicates that their algorithm uses $O(p^3M_t)$ additional control messages with the size of each being only $O(1)$, where p is the number of processes and M_t is the total number of truth value's changes of local predicates. In [6], Garg and Waldecker proposed an algorithm that evaluates the predicate in an *online* manner, i.e., predicates are evaluated immediately following each instruction's execution. This algorithm employs a central debugger to collect debug information from application processes and then performs the detection. The time complexity of the detection algorithm is $O(p^2m)$, where m is the maximum number of states in one application process. In comparison with Venkatesan and Dathan's algorithm, this algorithm uses only $O(M_r)$ additional control messages with the size of each being $O(p)$, where M_r is the total number of messages that all application processes receive.

One disadvantage of the above-mentioned algorithms is that the debugger evaluates execution states, which are collected from application processes, in a certain order. Restated, before evaluating certain states, all other states are queued. Since real systems can generate hundreds of states

- *L.-B. Chen is with the Department of Information Management, Chin-Min College, Tou-Fen, Miao-Li, Taiwan. E-mail: lbchen@csie.nctu.edu.tw.*
- *I.-C. Wu is with the Department of Computer Science and Information Engineering, National Chiao-Tung University, Hsin-Chu, Taiwan. E-mail: icwu@csie.nctu.edu.tw.*

Manuscript received 19 May 2000; revised 26 July 2001; accepted 22 Feb. 2002.
Recommended for acceptance by Luqi.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 112192.

0098-5589/02/$17.00 © 2002 IEEE

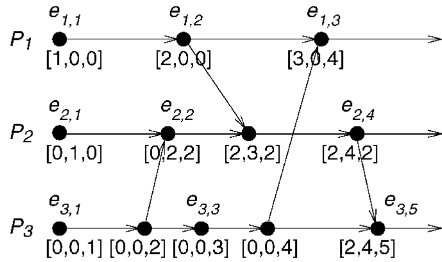


Fig. 1. Events and their time vectors.

in a very short time, the queues may grow unbounded. To solve this problem, in this paper, the notion of *removable states* is introduced. By discarding the removable states, the space requirement for each process can be minimized to $O(p)$, where p is the number of processes. While minimizing the memory space, time complexity is also improved to $O(pm)$ because the number of states that need to be evaluated is reduced. Our algorithm does not require the exchange of control messages during program execution because all the debug information is piggybacked in normal application messages.

The remainder of this paper is organized as follows: In Section 2, model and notations are defined. In Section 3, we shall introduce the notion of removable states and derive the condition of identifying removable states. Based on this result, Section 4 presents an efficient way to maintain nonremovable states and then discusses a new detection algorithm. Finally, a concluding remark is made in Section 5.

2 Model and Notations

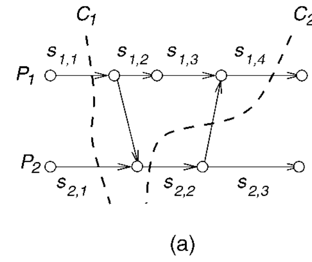
A distributed system consists of p *processes* denoted by $P_1, P_2, \ldots, P_p$. These processes share no global memory and no global clock. Message passing is the only way to communicate for processes. The transmission delay of the communication channel between each pair of processes is random. However, we assume that no message in any channel is lost, altered, or spuriously introduced.

2.1 States and Events

At a given time, the *state* of a process is defined by its variable's values. The *states* of processes can change only when *events* are executed. There are three kinds of events: an *internal event*, which performs a local computation, a *send event*, which sends a message to another process, and a *receive event*, which receives a message from another process via the channel.

The xth event occurring in process P_i is referred to as $e_{i,x}$. The number x is called the *sequence number* of $e_{i,x}$. Fig. 1 illustrates the events of the execution of a distributed program. Event $e_{i,x}$ *happens before* event $e_{j,y}$, denoted by $e_{i,x} \rightarrow e_{j,y}$, if and only if one of the following conditions holds [9]:

1. $i = j$ and $x < y$.
2. A message is sent from $e_{i,x}$ to $e_{j,y}$.
3. Another event $e_{k,z}$ exists such that $e_{i,x} \rightarrow e_{k,z}$ and $e_{k,z} \rightarrow e_{j,y}$.



(a)

Initial node
(1,1)
(2,1)
node (x,y)={$s_{1,x}$, $s_{2,y}$}
(2,2) (3,1)
(2,3) (3,2)
(3,3)
Final node
(4,3)

(b)

Fig. 2. (a) Space-time diagram of a distributed program. (b) The lattice of (a).

In this paper, the system is assumed to recognize the happened-before relationships by using *vector clocks* [10] (Fig. 1). With this approach, each process P_i maintains an integer vector $vector_i[1..p]$. Initially, each process P_i sets $vector_i$ to $[0, 0, \ldots, 0]$ and $vector_i[i] = 1$. When a process P_i executes an internal event, it increases $vector_i[i]$ by 1. When process P_i sends out a message, it increases $vector_i[i]$ by 1 and then associates $vector_i$ within the message. When process P_i receives a message associated with a vector, say v, it sets $vector_i[k] = max(vector_i[k], v[k]), \forall k$ and then increases $vector_i[i]$ by 1.

Let $vector(e_{i,x})$ represent the value of $vector_i$ after executing $e_{i,x}$ and before executing $e_{i,x+1}$. The following properties can be seen from Fig. 1: For event $e_{i,x}$, $vector(e_{i,x})[i] = x$ represents the sequence number of $e_{i,x}$ and $vector(e_{i,x})[j] = y$, $j \neq i$, represents the sequence number of event $e_{j,y}$, where $e_{j,y} \rightarrow e_{i,x}$ and $e_{j,y+1} \not\rightarrow e_{i,x}$. Therefore, the happened-before relationships can be determined in time $O(1)$ by using vector clocks, as shown in Theorem 1.

Theorem 1 ([10]). *For two events $e_{i,x}$ and $e_{j,y}$, $e_{i,x} \rightarrow e_{j,y}$ if and only if $vector(e_{i,x})[i] \leq vector(e_{j,y})[i]$.*

2.2 Global States and Global Predicates

A global state is a collection of states, one from each process, in which no happened-before relationship occurs. (Note that the system cannot enter a state with a happened-before relationship because messages cannot be received before they are sent.) For example, in Fig. 2a, C_1 is a global state, but C_2 is not. The set of all global states within a distributed program forms a *lattice* [3]. In the lattice, a node (global state) S_1 is linked to another node S_2 if the system can proceed from S_1 to S_2 by executing only one event. Fig. 2 shows the space-time diagram of a distributed program and the corresponding lattice. A possible run of a distributed

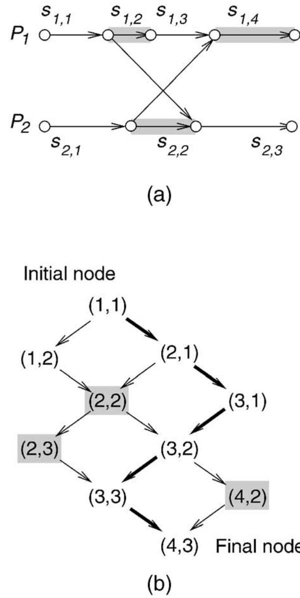


Fig. 3. An example of two-phase commit protocol. Let $\Phi = LP_1 \wedge LP_2$, where $LP_1 = \{P_1 \text{is committable}\}$ and $LP_2 = \{P_2 \text{is committable}\}$. (a) In process P_1 (P_2), a state is shaded if the local predicate LP_1 (LP_2) holds within this state. (b) A global state is shaded if Φ holds (i.e., both P_1 and P_2 are prepared to commit) within this global state.

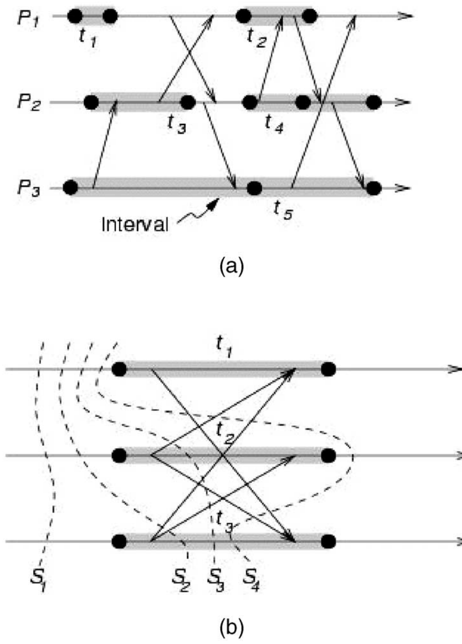


Fig. 4. (a) Example of intervals. (b) A scenario that Φ is not definitely true in intervals.

program can be viewed as a path in the lattice from the initial node (initial global state) to the final node (final global state). For example, the path depicted by bold lines in Fig. 2b represents a possible execution order of the events occurring in the program.

A *local predicate* is a Boolean expression of the process states. At any time, the process can evaluate its local predicate without communication. A global predicate is a Boolean expression, which involves the states of several processes. In this paper, we consider an important class of global predicates, known as *conjunctive predicate*, which can be expressed in a conjunctive form $LP_1 \wedge LP_1 \wedge \ldots \wedge LP_p$, where LP_i is the local predicate of process $P_i, i = 1, 2, ..., p$. For simplicity, we use either Φ or $LP_1 \wedge LP_1 \wedge \ldots \wedge LP_p$ to denote the conjunctive predicate.

In [8], Venkatesan and Dathan indicated that, in a typical software development environment, developers may have occasions to use the conjunctive predicate Φ in one or more of the following ways:

- $DEFINITELY(\Phi)$ is true if Φ is *definitely true*. Φ is *definitely true* if, for every path from the initial node to the final node in the lattice, Φ holds in some node. Detecting this kind of global predicate is usually used to ensure that a certain *desired* event would occur. For example, we can consider a distributed two-phase commit protocol (see Fig. 3). When the master decides to commit the transaction, it must assure that all the slaves are prepared to commit. Assume that there are two slaves, P_1 and P_2. Let $\Phi = LP_1 \wedge LP_2$, where $LP_1 = \{P_1 \text{ is committable}\}$ and $LP_2 = \{P_2 \text{ is committable}\}$. In Fig. 3b, a path (depicted by bold lines) exists in which Φ is not true in all nodes. Therefore, Φ is not definitely true. This implies that an error may occur because at least one slave process is not ready to commit during the program execution which corresponds to this boldface path.
- $POSSIBLY(\Phi)$ is true if Φ is *possibly true*. Φ is *possibly true* if a path exists in the lattice such that Φ holds in some node. Detecting this kind of predicate is usually used to ensure that certain *undesired* events would not occur. For example, consider a mutually exclusive program, which runs on a system with two processes P_1 and P_2. Let $\Phi = \{(P_1 \text{ is in the critical section})\} \wedge (P_2 \text{ is in the critical section})\}$. If Φ is possibly true, the undesired event (both processes are in the critical section) may occur in some run of the program.

2.3 Intervals

Researchers in [6], [8] proposed a necessary and sufficient condition of whether $DEFINITELY(\Phi)$ holds or not. This condition uses the notion of *intervals*. An interval t is a pair of events in the same process in which $t.lo$ and $t.hi$ are referred to as its *beginning event* and *ending event*, respectively. Furthermore, event $t.lo$ turns the truth value of the local predicate from false to true, events between $t.lo$ and $t.hi$ do not change the truth value, and event $t.hi$ turns the truth value from true to false. Two intervals, t and t', are *overlapped* if $t.lo \rightarrow t'.hi$ and $t'.lo \rightarrow t.hi$. For example, in Fig. 4a, interval t_2 and t_4 are overlapped, but t_2 and t_3 are not. A set of *overlapping global interval (OGI-set)* is a set of intervals, one from each process, in which each pair of intervals is overlapped. For example, interval set $\{t_2, t_4, t_5\}$ in Fig. 4a is an OGI-set.

To simplify, the following notations are defined for two interval sets I_1 and I_2:

- $I_1 = I_2$: For all $t \in I_1$ and $t' \in I_2$ in the same process, $t.lo = t'.lo$.

- $I_1 \preceq I_2$: For all $t \in I_1$ and $t' \in I_2$ in the same process, $t.lo \rightarrow t'.lo$ or $t.lo = t'.lo$. For example, in Fig. 4a, $\{t_1, t_3, t_5\} \preceq \{t_2, t_4, t_5\}$.
- $I_1 \rightarrow I_2$: For all $t \in I_1$ and $t' \in I_2$, $t.lo \rightarrow t'.hi$.

Fig. 4b illustrates an interesting scenario that $DEFINITELY(\Phi)$ does not hold. In this graph, t_1 and t_2 are not overlapped since there exists no message from $t_1.lo$ to $t_2.hi$. Thus, a program execution can be constructed such that Φ is true from global states S_1 to S_3, but it is false in S_4. Theorem 2 generalizes this scenario.

Theorem 2 [6], [8]. *For a distributed program, $DEFINITELY(\Phi)$ holds if and only if there exists an OGI-set.*

2.4 Distributed Online DEFINITELY Φ Detecting Problem

In a distributed environment, processes collect the execution states of other processes by exchanging messages. In other words, when a process P_i executes an event $e_{i,x}$, all the events (and the associated states) that P_i can observe are those that happen before $e_{i,x}$. These events are denoted by $E_{i,x}$, i.e., $E_{i,x} = \{e_{j,y} | e_{j,y} \rightarrow e_{i,x} \text{ or } e_{j,y} = e_{i,x}\}$. $E_{i,x}$ is called the *E-set* of $e_{i,x}$. If $E_{i,x} \subseteq E_{j,y}$, then $E_{j,y}$ is called a *future* E-set of $E_{i,x}$. The following property can be verified easily:

P1 Event $e_{i,x} \rightarrow e_{j,y}$ if and only if $E_{i,x} \subseteq E_{j,y}$.

In this paper, the distributed online $DEFINITELY(\Phi)$ detecting problem is that, whenever process P_i executes an event, say $e_{i,x}$, it tests whether or not $DEFINITELY(\Phi)$ holds for the debug information associated with E-set $E_{i,x}$.

3 Identifying Removable Intervals

According to Theorem 2, $DEFINITELY(\Phi)$ holds if and only if at least one OGI-set exists. To detect $DEFINITELY(\Phi)$ efficiently, the main idea of this paper is to derive the *minimum* OGI-set only and treat the others as *removable*. An OGI-set I in E-set $E_{i,x}$ is minimum if $I \preceq I'$ for all OGI-sets I' in $E_{i,x}$. The minimum OGI-set in $E_{i,x}$ is given by $F(E_{i,x})$.

To simplify our presentation, *pseudoevent* and *volatile interval* are defined in Definition 1.

Definition 1. *For an E-set $E_{i,x}$, the volatile interval $\widehat{t}$ for each process P_j is defined as follows:*

1. *If an interval t exists in P_j, it satisfies $t.lo \in E_{i,x}$ but $t.hi \notin E_{i,x}$ (e.g., interval t in Fig. 5), then $\widehat{t}.lo = t.lo$, and $\widehat{t}.hi$ is a pseudo event with $vector(\widehat{t}.hi) = [\infty, \infty, \ldots, \infty]$.*
2. *Otherwise, let $e_{j,y} \in E_{i,x}$ be the last event from P_j (e.g., event with vector clock $[3, 0, 0]$ in Fig. 5), both $\widehat{t}.lo$ and $\widehat{t}.hi$ are pseudo events where $vector(\widehat{t}.lo) = vector(e_{j,y})$, but $vector(\widehat{t}.lo)[j] = vector(e_{j,y})[j] + 1$ and $vector(\widehat{t}.hi) = [\infty, \infty, \ldots, \infty]$.*

The interval without any pseudoevents is called nonvolatile. *This E-set contains all events in $E_{i,x}$ and its pseudoevents are denoted by $\widehat{E}_{i,x}$. Notably, $E_{i,x} \subseteq \widehat{E}_{i,x}$. The vector of volatile intervals in $\widehat{E}_{i,x}$ is denoted by $f(\widehat{E}_{i,x})$.*

An E-set $E_{i,x}$ may not contain an OGI-set. However, $\widehat{E}_{i,x}$ always contains an OGI-set because intervals in $f(\widehat{E}_{i,x})$ are pairwisely overlapped. This is due to $vector(v.hi) = [\infty, \infty, \ldots, \infty]$ for any interval $v \in f(\widehat{E}_{i,x})$. Intuitively, $f(\widehat{E}_{i,x})$ is a candidate of OGI-sets within future E-sets. The following property is useful in the remainder of this paper:

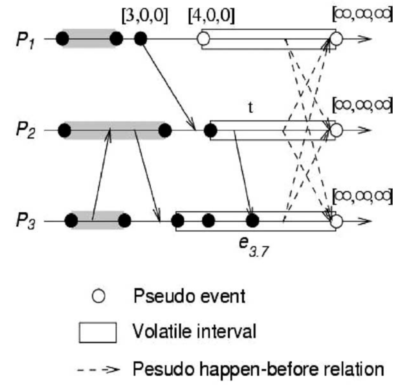


Fig. 5. Volatile intervals in E-set $E_{3,7}$.

$P2$ Let E-set $E_{i,x} = E_{j,y} \cup E_{k,z}$. The events in $E_{i,x}$ is from either $E_{j,y}$ or $E_{k,z}$. Hence, the interval t is nonvolatile in $E_{i,x}$ if and only if t is nonvolatile in either $E_{j,y}$ or $E_{k,z}$.

Given an E-set $E_{i,x}$, intervals are said to be removable if they do not belong to the minimum OGI-sets of any future E-set of $E_{i,x}$ because deriving the minimum is our only concern. Specifically, an interval $t \in E_{i,x}$ is $E_{i,x}$*-removable* if $t \notin F(E_{j,y})$ for all $E_{j,y}$, where $E_{i,x} \subseteq E_{j,y}$. Otherwise, t is $E_{i,x}$*-nonremovable*. Note that a removable interval must be nonvolatile. With this definition, the following property holds:

$P2$ If interval t is $E_{i,x}$*-removable*, then t is $E_{j,y}$*-removable* for all $E_{j,y}$ where $E_{i,x} \subseteq E_{j,y}$.

Next, a necessary and sufficient condition to identify removable intervals is derived in Theorem 3.

Theorem 3. *In an E-set $E_{i,x}$, a nonvolatile interval t is $E_{i,x}$-removable if and only if $t \notin F(\widehat{E}_{i,x})$.*

Proof. ($\Rightarrow$) If t is $E_{i,x}$-removable, since $\widehat{E}_{i,x}$ is a future E-set of $E_{i,x}$, $t \notin F(\widehat{E}_{i,x})$ can be derived (Property P3).

($\Leftarrow$) Let $E_{j,y}$ be a future E-set of $E_{i,x}$. We shall prove this direction by showing that if $t \in F(\widehat{E}_{j,y})$, then $t \in F(\widehat{E}_{i,x})$. As illustrated in Fig. 6, partition $F(\widehat{E}_{j,y})$ into $S_1 \cup S_2$, where $S_2 \subseteq \widehat{E}_{i,x}$ and $S_1 = F(\widehat{E}_{j,y}) \setminus S_2$ (note that $S_2 \neq \{\}$ since $x \in S_2$). Since $S_1 \rightarrow t$, we can derive that there exists a set of volatile intervals in $\widehat{E}_{i,x}$, say S_1', such that the sets of the beginning events of S_1 and S_1' are identical.

The set $S_1' \cup S_2$ is an OGI-set in $\widehat{E}_{i,x}$ since $S_1' \rightarrow S_2$ and $S_2 \rightarrow S_1'$ (because the intervals in S_1' are volatile). Next, we show that $S_1' \cup S_2$ is a minimum OGI-set of $\widehat{E}_{i,x}$, i.e., $S_1' \cup S_2 = F(\widehat{E}_{i,x})$. By contradiction, assume that $S_1' \cup S_2 \preceq F(\widehat{E}_{i,x})$ and $S_1' \cup S_2 \neq F(\widehat{E}_{i,x})$. As shown in Fig. 6, partition $F(\widehat{E}_{i,x})$ into $T_1' \cup T_2$, where T_1' and T_2 are sets with volatile and nonvolatile intervals, respectively. Through Fig. 6, we can derive that $T_1' \subseteq S_1'$ because $\{T_1' \cup T_2\} \preceq \{S_1' \cup S_2\}$ and a volatile interval must be the last interval in the process. Let T_1 represent the set containing nonvolatile intervals in S_1, as shown in Fig. 6.

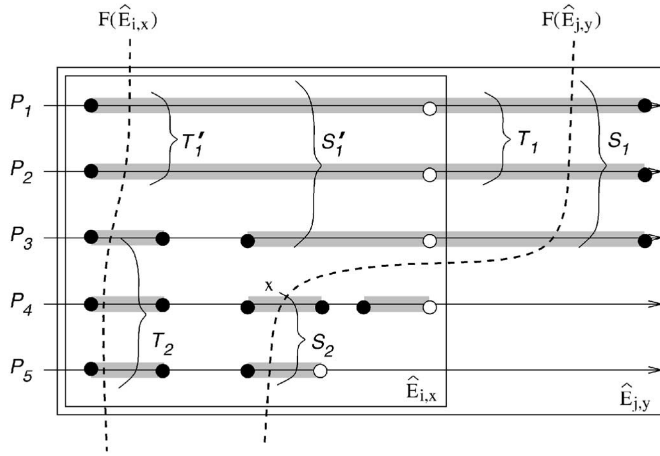


Fig. 6. Illustration of the minimum OGI-sets of E-set $\widehat{E}_{i,x}$ and $\widehat{E}_{j,y}$, where $E_{i,x} \subset E_{j,y}$. Note that the messages are not drawn for clarity in this figure.

(Note that the set of beginning events of T_1 and T_1' are identical.) Then, $T_1 \cup T_2$ is an OGI-set in $E_{j,y}$ because:

1. $T_1 \rightarrow T_2$ in $E_{j,y}$ since $T_1' \rightarrow T_2$ in $F(\widehat{E}_{i,x})$.
2. $T_2 \rightarrow T_1$ in $E_{j,y}$ since $T_2 \rightarrow S_2$, $S_2 \rightarrow S_1$, and $T_1 \subseteq S_1$.

Therefore, $T_1 \cup T_2$ is an OGI-set in $E_{j,y}$ and $\{T_1 \cup T_2\} \preceq \{S_1 \cup S_2\}$. This derivation contradicts with the fact that $S_1 \cup S_2$ is the minimum OGI-set in $E_{j,y}$. Therefore, the theorem holds. □

The following corollary extended from this theorem is useful for the remaining part of this paper.

Corollary 1. *If E-set $E_{j,y} \subseteq E_{i,x}$, then $F(\widehat{E}_{j,y}) \preceq F(\widehat{E}_{i,x})$.*

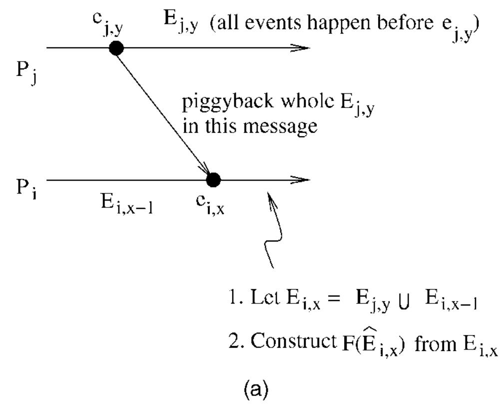
Proof. Based on Theorem 3, this corollary is accurate because removable intervals in the E-set $E_{j,y}$ must also be removable in its future E-sets. □

4 Distributed Online Algorithm to Detect $DEFINITELY(\Phi)$

4.1 Using the Notion of Removable States

This section shows the difference between the detection algorithms with and without using the notion of removable states. Fig. 7 illustrates the detection scenarios. In Fig. 7a, when process P_i receives a message by executing event $e_{i,x}$, it first constructs the new E-set $E_{i,x}$ from its old E-set $E_{i,x-1}$ and the newly received E-set $E_{j,y}$, i.e., $E_{i,x} = E_{i,x-1} \cup E_{j,y}$. Then, it computes the set $F(\widehat{E}_{i,x})$ from the set $E_{i,x}$. Unfortunately, this simple approach is intractable since the E-sets grow each time as an instruction is executed. Part b of Fig. 7b illustrates how the notion of removability can be applied to improve the detection. It employs the minimum OGI-set only rather than employing the whole E-set, based on the concept depicted in Corollary 2. Clearly, this approach incurs a very low overhead to the distributed system because an OGI-set contains $O(p)$ members only.

Corollary 2. *Assume that process P_j sends a message, say m, to process P_j, where the send event and receive event of m are $e_{j,y}$ and $e_{i,x}$, respectively. The intervals in $F(\widehat{E}_{i,x})$ must be from either $F(\widehat{E}_{i,x-1})$ or $F(\widehat{E}_{j,y})$.*



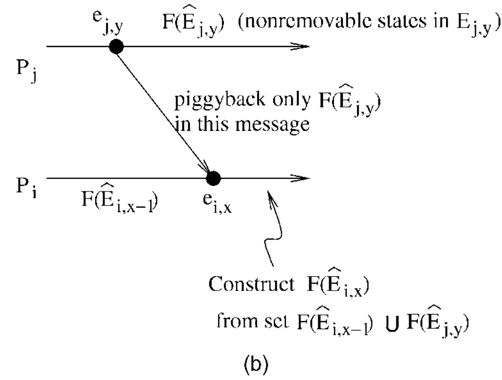


Fig. 7. (a) Illustration of the detection without the notion of removable states. (b) Illustration of the detection by discarding the removable states.

Proof. E-set $E_{i,x}$ is a future E-set of $E_{i,x-1}$ and $E_{j,y}$ since $E_{i,x} = E_{i,x-1} \cup E_{j,y}$. According to Theorem 3, the intervals in $F(\widehat{E}_{i,x})$ must be from either $F(\widehat{E}_{i,x-1})$ or $F(\widehat{E}_{j,y})$. □

A naive approach to computing $F(\widehat{E}_{i,x})$ in Fig. 7b is to apply the algorithms that were proposed in [6], [8] and let $F(\widehat{E}_{i,x-1})$ and $F(\widehat{E}_{j,y})$ be the inputs of the algorithms. Their algorithms are operated by testing overlap between intervals and by removing useless intervals systematically. However, in a worst case, in each run, it performs $O(p^2)$ testing to ensure that all of the p intervals (one from each process) are pairwise overlapped. The total time is $O(p^2m)$ if the maximum number of events in one process is m. In this section, a more efficient detection algorithm that runs in time $O(pm)$ is proposed.

In Section 4.2, we present an efficient approach to maintain the minimum OGI-sets. Based on this result, Section 4.3 presents our new detection algorithm and its complexity and correctness are analyzed.

4.2 Maintain Minimum OGI-Sets Efficiently

Let X, Y, and Z be the E-sets satisfying the condition $Z = X \cup Y$. This section describes how to derive $F(Z)$ from $F(X)$ and $F(Y)$. Before describing our approach, some notations used in this section are defined as follows: To identify one interval t in different E-sets, let $t^{(A)}$ refer to t in

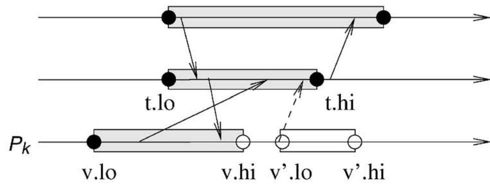


Fig. 8. A volatile interval v becomes removable implies that all of the nonvolatile intervals become removable.

E-set A (i.e., $t.lo = t^{(A)}.lo$). An interval $t^{(A)}$ is said to be B-*removable* if $t^{(B)}$ exists in B and is B-*removable*.

The minimum OGI-set of Z can be derived by finding that those intervals not removable in X or Y, but become removable in Z $(= X \cup Y)$. The following Lemma demonstrates that the nonvolatile intervals remain nonremovable until some volatile interval becomes removable.

Lemma 1. *Let $Z = X \cup Y$ be the E-sets as described above. All the nonvolatile intervals in $F(\widehat{X})$ become Z-removable if and only if some volatile interval in $F(\widehat{X})$ becomes Z-removable.*

Proof. $(\Leftarrow)$ Consider a nonvolatile interval $t^{(X)}$ and a volatile interval $v^{(X)}$ in $F(\widehat{X})$. Assume that $v^{(X)}$ is in process P_k. Fig. 8a illustrates that v is the last interval in P_k that can overlap with t. (Otherwise, an interval v' exists in P_k such that $v.hi \rightarrow v'.lo \rightarrow t.hi$; this contradicts the condition that v is volatile in X.) This implies that, if v becomes Z-*removable*, then t also becomes Z-*removable*.

$(\Rightarrow)$ By contradiction, assume that all volatiles in $F(\widehat{X})$ are Z-*nonremovable* whereas the nonvolatile t is Z-*removable*. $F(\widehat{X}) = F(\widehat{Z})$ is proven as follows: For each ordered pair of $t_1, t_2 \in F(\widehat{X})$ (note that $t_1^{(X)} \rightarrow t_2^{(X)}$), the property $t_1^{(Z)} \rightarrow t_2^{(Z)}$ holds because:

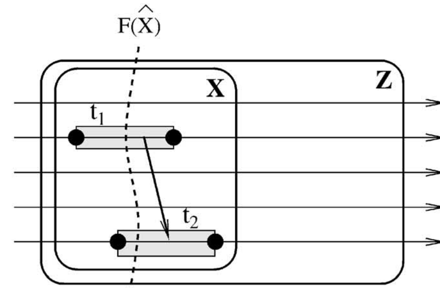
- If $t_2^{(X)}$ is nonvolatile (Fig. 9a), then $t_1^{(X)} \rightarrow t_2^{(X)}$ implies $t_1^{(Z)} \rightarrow t_2^{(Z)}$.
- If $t_2^{(X)}$ is volatile and $t_2^{(Z)}$ remains volatile (Fig. 9b), clearly, $t_1^{(Z)} \rightarrow t_2^{(Z)}$ (see Definition 1).
- If $t_2^{(X)}$ is volatile but $t_2^{(Z)}$ becomes nonvolatile (Fig. 9c), the property $t_1^{(Z)} \rightarrow t_2^{(Z)}$ is satisfied because: $t_1^{(X)} \in F(\widehat{X})$, $t_2^{(Z)} \in F(\widehat{Z})$ (this is because $t_2^{(X)}$ is volatile in $F(\widehat{X})$ and, thus, is Z-*nonremovable* by assumption), and $F(\widehat{X}) \preceq F(\widehat{Z})$ (see Corollary 1).

Therefore, $F(\widehat{X}) = F(\widehat{Z})$, contradicts with the fact that $t \in F(\widehat{Z})$ is Z-removable. □

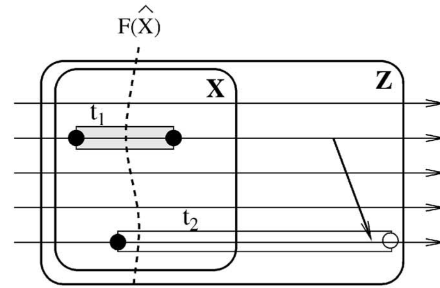
Next, Lemmas 2 and 3 demonstrate the condition for volatile intervals.

Lemma 2. *Let $Z = X \cup Y$ be the E-sets as described above. If neither $F(\widehat{X}) \preceq F(\widehat{Y})$ nor $F(\widehat{Y}) \preceq F(\widehat{X})$, then, $F(\widehat{Z}) = f(\widehat{Z})$.*

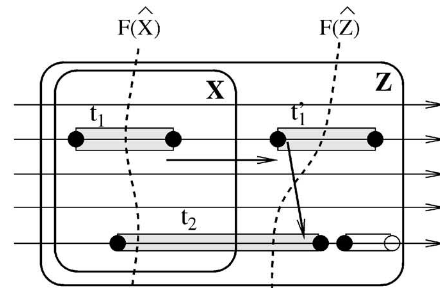
Proof. If $F(\widehat{X}) \npreceq F(\widehat{Y})$ and $F(\widehat{Y}) \npreceq F(\widehat{X})$, there exist intervals $t_1, t_2 \in F(\widehat{X})$ and $u_1, u_2 \in F(\widehat{Y})$ such that $\{t_1\} \prec \{u_2\}$ and $\{u_1\} \prec \{t_2\}$. Therefore, t_1 and u_1 are nonvolatile in Z and are Z-*removable*. From Lemma 1, all nonvolatile intervals in $F(\widehat{X})$ and $F(\widehat{Y})$ are Z-removable. Thus, $F(\widehat{Z}) = f(\widehat{Z})$. □



(a)



(b)



(c)

Fig. 9. Illustration of proof of Lemma 1.

Lemma 3. *Let $Z = X \cup Y$ be the E-sets as described above. Furthermore, assume that $F(\widehat{Y}) \preceq F(\widehat{X})$. A volatile interval $t^{(X)} \in F(\widehat{X})$ becomes Z-removable if and only if $t^{(Y)}$ is nonvolatile and the following properties are satisfied:*

1. *$t^{(Y)}$ is Y-removable, or*
2. *$t^{(Y)}$ is Y-nonremovable and some volatile interval $u \in F(\widehat{Y})$ is X-removable.*

Proof. In this proof, only the second case, i.e., t is Y-*nonremovable*, is considered. (In the first case, clearly, t is Y-*removable* implies that t is Z-*removable*.)

$(\Leftarrow)$ If u is X-*removable*, from Lemma 1, all the nonvolatile intervals in $F(\widehat{Y})$, including $t^{(Y)}$, are Z-*removable*.

$(\Rightarrow)$ First, $t^{(Y)}$ must be nonvolatile from Property P2. By contradiction, assume that t is Z-*removable* but $t^{(Y)}$ is Y-*nonremovable* and all the volatile intervals $u \in F(\widehat{Y})$ are X-*nonremovable*. This assumption implies that $F(\widehat{X}) = F(\widehat{Y})$ (the detail is ignored since the proof is very similar to Lemma 1). Since $Z = X \cup Y$, $F(\widehat{X}) = F(\widehat{Y})$ is an OGI-set in

Z. This implies that $F(\widehat{Z}) = F(\widehat{X}) = F(\widehat{Y})$, contradicting with the fact that $t \in F(\widehat{X}) = F(\widehat{Z})$ is *Z-removable*. □

4.3 The New Detection Algorithm

In our new detection algorithm, each process only keeps two *p-tuple* vectors F and f to represent its minimum OGI-set and its volatile interval set, respectively. This algorithm consists of the following procedures that are executed at each process P_i:

- **Procedure** InternalEvent (to be called when P_i executes an internal event, say $e_{i,x}$):
  1. After execution of event $e_{i,x}$, the truth value of LP_i may change, as follows:
     a. The truth value of LP_i is unchanged: Both F and f remain unchanged.
     b. The truth value of LP_i is in a false-to-true transition: Modify $f[i]$ to the new volatile interval with its beginning event being $e_{i,x}$.
     c. The truth value of LP_i is in a true-to-false transition: In this case, the current volatile interval of process P_i is ended at event $e_{i,x}$. Modify $f[i]$ to the new volatile interval in which both beginning and end events are pseudo.
- **Procedure** SendEvent (to be called when P_i executes a send event, say $e_{i,x}$):
  1. Since the truth value of LP_i is unchanged, both F and f are also unchanged.
  2. Piggyback F and f in the message and then send it.
- **Procedure** ReceiveEvent (to be called when P_i executes a receive event, say $e_{i,x}$):
  1. Assume that the sent event of this message is $e_{j,y}$. Receive the message and extract data F' (represent $F(\widehat{E}_{j,y})$) and f' (represent $f(\widehat{E}_{j,y})$).
  2. Let $f[k] = \max(f[k], f'[k])$, $k = 1, 2, \ldots, p$.
  3. (Based on Lemma 2) If $F(\widehat{E}_{i,x-1}) \not\preceq F(\widehat{E}_{j,y})$ or $F(\widehat{E}_{j,y}) \not\preceq F(\widehat{E}_{i,x-1})$, then $F = f$. Go to Step 7.
  4. If $F(\widehat{E}_{j,y}) \preceq F(\widehat{E}_{i,x-1})$, determine whether an interval t is $E_{i,x}$*-removable* as follows:
     a. (Based on Property P3) If $t \prec F(\widehat{E}_{i,x-1})$ then t is removable.
     b. (Based on Lemma 3) If $t \in F(\widehat{E}_{i,x-1})$ and t is volatile:
        i. If the following properties are satisfied, mark t as removable:
           A. t is $E_{j,y}$*-removable*, or
           B. t is $E_{j,y}$*-nonremovable* and some volatile interval $u \in F(\widehat{E}_{j,y})$ is $E_{i,x-1}$*-removable*. For example, consider an interval t_2 illustrated in Fig. 10. The interval t_2 is $E_{j,y}$-nonremovable and volatile interval $t_1 \in F(\widehat{E}_{j,y})$ is $E_{i,x-1}$-removable (Fig. 10a and Fig. 10b). Hence, t_2 is $E_{i,x}$-removable (Fig. 10c).
     c. (Based on Lemma 1) If $t \in F(\widehat{E}_{i,x-1})$ and t is nonvolatile:
        i. If any volatile interval is marked as removable in step 4b, mark t as removable.
  5. If $F(\widehat{E}_{i,x-1}) \preceq F(\widehat{E}_{j,y})$, repeat Step 4 except that the roles of $F(\widehat{E}_{i,x-1})$ and $F(\widehat{E}_{j,y})$ are swapped.
  6. Let $F(\widehat{E}_{i,x})$ be the set of intervals which have not been marked as removable in previous steps.
  7. If $F(\widehat{E}_{i,x})$ contains no volatile, then $DEFINITELY(\Phi)$ is true. Otherwise, $DEFINITELY(\Phi)$ is false.

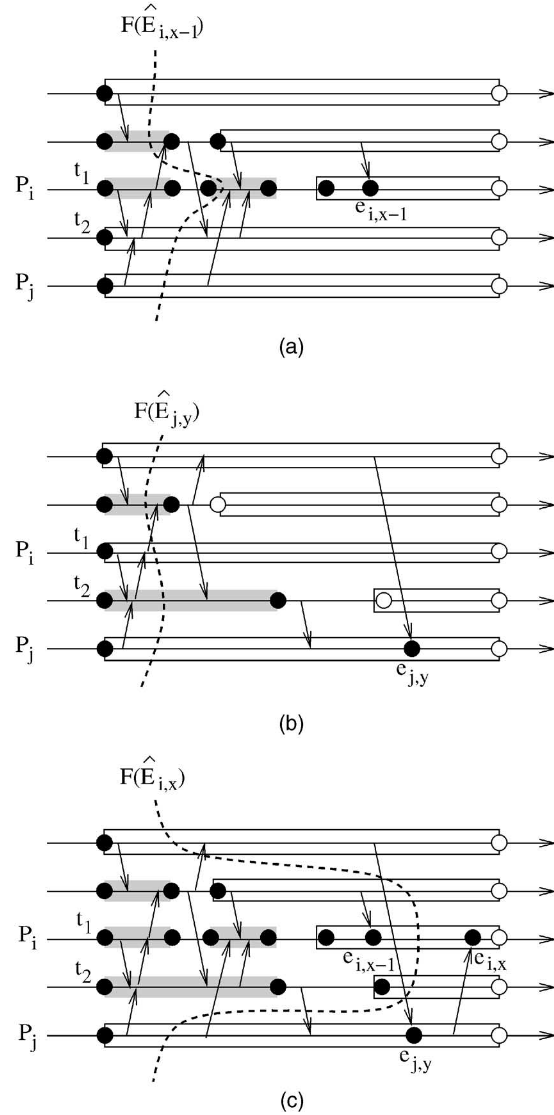


Fig. 10. Illustration of (a) $F(\widehat{E}_{i,x-1})$, (b) $F(\widehat{E}_{j,y})$, and (c) $F(\widehat{E}_{i,x})$.

In this algorithm, a process cannot evaluate the global predicate if there are no messages sent from other processes to carry the debug information. To solve this problem, if $DEFINITELY(\Phi)$ is still false at the end of the program execution, p extra messages are sent among all p processes in a circular way to pass the debug information. However,

as compared with the cost of the entire distributed computation, these p messages incur a very low overhead.

The correctness of this algorithm can be verified easily based on the theorems presented in Section 4.2. Before analyzing the complexity of the algorithm, the implementation has to be explained. First, vectors F and f are implemented by using vectors of integers: When process P_i executes the event $e_{i,x}$, the value $F[j]$ (resp. $f[j]$) equals the sequence number of the beginning event of interval $F(\widehat{E}_{i,x})[j]$ (resp. $f(\widehat{E}_{i,x})[j]$). The operations of the algorithm is implemented as follows:

- $F(\widehat{E}_{j,y}) \preceq F(\widehat{E}_{i,x-1})$ if and only if $F'[k] \leq F[k]$ for all k (assume that F' refers to $F(\widehat{E}_{j,y})$ and F refers to $F(\widehat{E}_{i,x-1})$). This operation takes $O(p)$ time.
- Interval $F[k]$ is volatile if and only if $F[k] = f[k]$. This operation takes $O(1)$ time.
- Interval t with $t.lo = e_{k,z}$ is $E_{i,x}$-*removable* if t is nonvolatile and $F[k] \neq z$, (assume that F refers to $F(\widehat{E}_{i,x})$). This operation takes $O(1)$ time.

Based on the above implementation, each invocation of the procedures (InternalEvent, SendEvent, and ReceiveEvent) requires $O(p)$ time. Assuming that there are m_i events for process P_i, the total time complexity for the process is $O(pm_i)$.

5 Discussion

This paper investigates the problem of detecting the definitely true conjunctive predicates ($DEFINITELY(\Phi)$). To solve the problem of unbounded queue growth resulting from previous algorithms, in this paper, the notion of removable states is introduced. By discarding the removable states, the space requirement for each process can be minimized to $O(p)$, where p is the number of processes. While bounding the memory space, this analysis shows that the time complexity of the proposed algorithm is only $O(pm)$, which is faster than previous algorithms by a factor of p.

Another related detection problem regarding the distributed debugging is to detect whether the conjunctive predicates are possibly true ($POSSIBLY(\Phi)$) [5], [7], [8]. To enhance the performance of $POSSIBLY(\Phi)$ detection algorithms, Chiou and Korfhage [11] presented two algorithms to remove some useless states for the detection. However, their algorithms run in a centralized environment and identify the partial useless states only. For the distributed detection of $POSSIBLY(\Phi)$, finding an efficient approach to identify the removable states is still a research topic.

Acknowledgments

The authors would like to thank the National Science Council of the Republic of China for financially supporting this research under contract No. NSC-89-2213-E-243-001. The authors would also like to thank the anonymous referees for their valuable comments, which greatly improved the presentation of this paper.

References

[1] R. Copper and K. Marzullo, "Consistent Detection of Global Predicates," *Sigplan Notices,* pp. 167-174, 1991.
[2] K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans. Computing Systems,* vol. 3, no. 1, pp. 63-75, Feb. 1985.
[3] R. Cooper and K. Marzullo, "Consistent Detection of Global Predicates," *SIGPLAN Notices,* pp. 167-174, 1991.
[4] C.M. Chase and V. K. Garg, "Efficient Detection of Restricted Classes of Global Predicates," *Proc. Ninth Int'l Workshop Distributed Algorithms,* Sept. 1995.
[5] V.K. Garg and B. Waldecker, "Detection of Weak Unstable Predicates in Distributed Programs," *IEEE Trans. Parallel and Distributed Systems,* vol. 5, no. 3, pp. 299-307, March 1994.
[6] V.K. 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, Dec. 1996.
[7] M. Hurfin, M. Mizuno, M. Raynal, and M. Singhal, "Efficient Distributed Detection of Conjunctions of Local Predicates," *IEEE Trans. Software Eng.,* vol. 24, no. 8, pp. 664-677, Aug. 1998.
[8] S. Venkatesan and B. Dathan, "Testing and Debugging Distributed Systems Using Global Predicates," *IEEE Trans. Software Eng.,* vol. 21, no. 2, pp. 165-169, Feb. 1995.
[9] L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," *Comm. ACM,* vol. 21, no. 7, pp. 558-565, July 1978.
[10] F. Mattern, "Virtual Time and Global States of Distributed Systems," *Parallel and Distributed Algorithms: Proc. Int'l. Workshop Parallel and Distributed Algorithms,* pp. 215-226, 1988.
[11] H.K. Chiou and W. Korfhage, "Enhancing Distributed Event Predicate Detection Algorithms," *IEEE Trans. Parallel and Distributed Systems,* vol. 7, no. 7, pp. 673-676, July 1996
[12] P.Y. Chung, Y.M. Wang, and I.J. Lin, "Checkpoint Space Reclamation for Uncoordinated Checkpointing in Message-Passing Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 8, no. 6, pp. 165-169, June 1997.

Loon-Been Chen received the BS degree from Soochow University in September 1991, the MS degree from National. Chung-Cheng University in September 1993, and the PhD degree from National-Chiao-Tung University in January 1999, all in computer science. Currently, he is an assistant professor of information management at the Chin-Min College, Miao-Li, Taiwan. His research interests include distributed computing, internet/network computing, and XML documents processing.

I-Chen Wu received the BS degree in electrical engineering from National Taiwan University in 1982, the MS degree in computer science from National Taiwan University in 1984, and the PhD degree in computer science from Carnegie Mellon University in 1993. He is currently an associate professor in the Department of Computer Science and Information Engineering of National Chiao-Tung University, Taiwan. His research interests include internet computing, distributed computing, and software engineering.

▷ **For more information on this or any computing topic. please visit our Digital Library at** http://computer.org/publications/dilb.