Precluding Useless Events for On-Line Global Predicate Detections

Li-Hsing Yen
Department of Computer Science and Information Engineering
Chung Hua University
Hsinchu, Taiwan 30067, R.O.C.
lhyen@chu.edu.tw

Abstract

Detecting global predicates is an important task in testing and debugging distributed programs. In this paper, we propose an approach that effectively precludes useless events for global predicate detection, facilitating the process of an independent on-line checking routine. To identify more useless events than a simple causality-check method can do, our method tracks and maintains the precedence information of event intervals as a graph. To reduce the potentially expensive space and time cost as the graph expands, we propose an effective scheme to prune the graph. The performance of our method is analyzed and evaluated by simulations. The result shows that our approach outperforms conventional approaches in terms of the number of useless events found.

1. Introduction

One important task in testing and debugging a distributed program is to answer whether a given execution run of this program fulfills a particular property. Such a property is often specified as a global predicate—a Boolean expression whose value depends on the state of multiple processes and, perhaps, communication channels. Detecting global predicates involves identifying consistent global states [2], on which predicates are to be evaluated. In general, the number of consistent global states is exponential in the number of processes [5]. Therefore an exhaustive search for all system states, which is necessary for detecting a general-form predicate, will suffer from the combinatorial explosion problem. Many researchers circumvent this problem by placing restriction on the types of predicates. In particular, they have considered global predicates that can be logically decomposed into sub-expressions, each of which is locally detectable by a single process [13, 11, 3, 8, 15, 9]. Such sub-predicates are called local predicates. Only local states upon which local predicates are satisfied have to be examined to see if any combination of them can form a consistent global state. The number of states examined is therefore reduced.

A typical procedure toward this kind of predicate detection is as follows [3, 8, 9]. Consider a distributed system that consists of n processes, labeled $P_1, P_2, \ldots, P_n$. All processes cooperatively maintain vector clocks [6, 12] which are used to timestamp events. This timestamping scheme possesses a desired property that for any two events a and b, we can determine if a *happens before* [10] b (denoted by $a \rightarrow b$) by comparing their timestamps. Once an event upon which a local predicate is evaluated as *true* occurs, the process sends in FIFO order the event's identification together with its timestamp to a dedicated *checker* process [8, 3]. The checker process maintains n event queues, $Q_1, Q_2, \ldots, Q_n$, where each Q_i is for storing events from P_i. Events in each queue are arranged in the order as they occurred: the head event of Q_i is the earliest predicate event among other events occurred in P_i.

The checker process performs two routines. The first routine examines if the event set comprising all current head events in each queue is consistent. The technique used in this routine is to repeatedly find two causally related head events and remove the one that happened before the other [7, 8]. Whenever no event queue is empty and the checker cannot remove any head event, a consistent global predicate is identified.

Events usually do not arrive at a constant rate, so it is possible that some event queue grows lengthily while others drain. If any event queue is empty, the first routine must wait for all absent events before it can proceed. Checker's second routine identifies and removes all events currently pending in queues that are evidently inconsistent with other not yet arrived events. Precluding *useless events* not only reduces space requirement on event queues, but also avoids further process of these events by the first routine. This removal does not impose additional computation overhead on the checker process: while the first routine cannot progress, the checker process can perform this routine rather than be-

ing idle. In this paper, we focus on the design of this routine.

Suppose that k event queues are non-empty, each of which has m events. A naive approach might involve looking at all m^k possible sets consisting of one event from each of the k event queues [4]. Chiou and Korfhage [4] proposed an algorithm for finding removable events that has an $O(k^2m^2)$ time complexity. Their method, however, cannot identify all useless events. Although the necessary and sufficient conditions for useless events have been formulated [14, 1], as to the author's knowledge, no practical algorithm has been proposed. Our method exploits the result in [1], which established the theory of event intervals, and treats the problem of finding useless events as an on-line computation of a reachability matrix representing an event interval graph. We prevent unlimited expansion of the matrix by cutting down obsolete rows and columns, saving both memory space and execution time. The validity proof of our method is provided. The simulation result shows that our approach outperforms conventional approaches in the number of useless events found.

2. Preliminary

2.1. Definitions

An event upon which a local predicate is evaluated as *true* is defined as a predicate event. Let e_i^j denote the j-th predicate event occurred at P_i. Event interval $\mathcal{I}_i^j$ is the set of all events between e_i^j and e_i^{j+1}, including e_i^j but excluding e_i^{j+1}.[1] We define *precedence relation* between event intervals as follows.

Definition 1 The direct precedence relation ($\prec_d$). Let $\mathcal{I}_i^x$ and $\mathcal{I}_j^y$ be two event intervals. $\mathcal{I}_j^y \prec_d \mathcal{I}_i^x$ if and only if (1) $j = i$ and $x = y + 1$ or (2) there exists a message that is sent from some event in $\mathcal{I}_j^y$ and is received by some event in $\mathcal{I}_i^x$.

Definition 2 The precedence relation (denoted by $\prec$) is the transitive closure of $\prec_d$.

We say that $\mathcal{I}_j^y$ *immediately precedes* $\mathcal{I}_i^x$ if $\mathcal{I}_j^y \prec_d \mathcal{I}_i^x$, and that $\mathcal{I}_j^y$ *precedes* $\mathcal{I}_i^x$ if $\mathcal{I}_j^y \prec \mathcal{I}_i^x$. A *precedence graph* is a directed graph $(\mathcal{I}, E)$, where $\mathcal{I}$ is a set of event intervals, and $\langle \mathcal{I}_i^x, \mathcal{I}_j^y \rangle \in E$ iff $\mathcal{I}_i^x \prec_d \mathcal{I}_j^y$. Figure 1 (b) shows the precedence graph corresponding to the sample execution run in Figure 1 (a).

2.2. Previous work

Netzer and Xu [14] showed that an event cannot be in any consistent event set if and only if there is a zigzag path from this event to itself (called a *zigzag cycle*). Formally, a zigzag path exists from e_i^p to e_j^q iff there are messages $m_1, m_2, \ldots, m_n$ $(n \geq 1)$ such that (1) m_1 is sent by P_i after the occurrence of e_i^p, (2) if $m_r (1 \leq r < n)$ is received by P_k, then m_{r+1} is sent by P_k in the same or later event interval, and (3) m_n is received by P_j before e_j^q occurs. The concept of zigzag cycles provides a theoretical basis for identifying removable events. However, it is difficult to capture zigzag paths by means of any causality tracking scheme such as vector clocks, since a zigzag path is not necessarily a causal path. In the past, only certain types of zigzag paths and thus only some sort of removable events could be effectively identified. In Chiou and Korfhage's method [4], only those events e_i^k for which $\exists j, l : e_j^l \rightarrow e_i^k \wedge e_i^k \rightarrow e_j^{l+1}$ can be identified. Their method therefore deals with only zigzag cycles that involve exactly two processes and two messages. Given the sample execution shown in Figure 1 (a), their method can detect only useless event e_1^1. Event e_2^2, although useless, is not detectable.

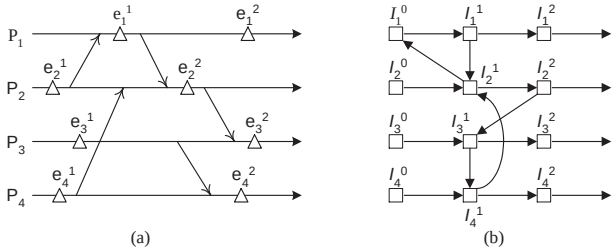

Figure 1. (a) Sample execution run of a four-process system; (b) Precedence graph corresponding to (a)

Baldoni et. al. [1] viewed the same problem from a different aspect. They showed that a predicate event e_i^k is removable (impossible to be in any consistent event set) if and only if $\mathcal{I}_i^k \prec \mathcal{I}_i^{k-1}$, which is consistent with Netzer and Xu's result but focus on the relation between event intervals rather than events. Their results motivated our research. Instead of identifying zigzag paths between predicate events, our proposed method tracks precedence relation between event intervals.

2.3. Theoretical basis and design issues

The idea behind our approach is that finding removable events can be transformed to the problem of detecting cycles in the precedence graph, as stated below. Each non-checker process tracks all event intervals that immediately precede its current one, and sends this information to the checker process when the current event interval is completed. Based on information sent from non-checker processes, the checker process constructs a precedence graph.

[1] For completeness, $\mathcal{I}_i^0$ denotes the sequence of events between P_i's first event (not necessarily a predicate event) and e_i^1.

Whether $\mathcal{I}_i^k \prec \mathcal{I}_i^{k-1}$ for any two successive event intervals $\mathcal{I}_i^{k-1}$ and $\mathcal{I}_i^k$ can then be determined by checking the existence of a path from $\mathcal{I}_i^k$ to $\mathcal{I}_i^{k-1}$ in the precedence graph.

However, when turning this conceptual result into a practical on-line detection algorithm, we may encounter some difficulties. First, the checker process can construct only a subgraph of the complete precedence graph, which we define as *constructed graph*.

Definition 3 A constructed graph $G_c = (\mathcal{I}_c, E_c)$ is a subgraph of a precedence graph $G = (\mathcal{I}, E)$ in which the vertices are partitioned into two disjoint sets R and C such that

- for every $\mathcal{I}_i^x \in \mathcal{I}$ and $\mathcal{I}_j^y \in \mathcal{I}$, if $\langle \mathcal{I}_j^y, \mathcal{I}_i^x \rangle \in E_c$, then $\mathcal{I}_i^x \in C$, and
- for every $\mathcal{I}_i^x \in C$ and $\mathcal{I}_j^y \in \mathcal{I}$, if $\langle \mathcal{I}_j^y, \mathcal{I}_i^x \rangle \in E$, then $\mathcal{I}_j^y \in \mathcal{I}_c$ and $\langle \mathcal{I}_j^y, \mathcal{I}_i^x \rangle \in E_c$

Intuitively, an event interval is in C if all event intervals immediately preceding it, if any, are either in C or in R. An event interval is in R if it is not in C and it immediately precedes at least one event interval in C.

Constructed graph offers only partial information about precedence relation. For any two vertices $\mathcal{I}_i^x$ and $\mathcal{I}_j^y$ in a constructed graph, we have $\mathcal{I}_j^y \prec \mathcal{I}_i^x$ if there exists a path from $\mathcal{I}_j^y$ to $\mathcal{I}_i^x$, but the reverse implication does not hold. Therefore, with a naive and straightforward algorithm design, we may not reach a conclusion until the execution of the monitored program is completed, at which time the entire precedence graph is obtained. We overcame this difficulty by observing that the precedence relation is complete for all vertices in a particular subgraph of the precedence graph. This subgraph is defined as *explored graph*.

Definition 4 An explored graph $G_e = (\mathcal{I}_e, E_e)$ is a subgraph of a precedence graph $G = (\mathcal{I}, E)$ with the property that for every $\mathcal{I}_i^x \in \mathcal{I}_e$ and $\mathcal{I}_j^y \in \mathcal{I}$, if $\langle \mathcal{I}_j^y, \mathcal{I}_i^x \rangle \in E$, then $\mathcal{I}_j^y \in \mathcal{I}_e$ and $\langle \mathcal{I}_j^y, \mathcal{I}_i^x \rangle \in E_e$

Lemma 1 An explored graph is a constructed graph with the property that $R = \emptyset$.

Theorem 1 Let $\mathcal{I}_i^x$ and $\mathcal{I}_j^y$ be any two vertices in an explored graph. We have $\mathcal{I}_j^y \prec \mathcal{I}_i^x$ iff there exists a path from $\mathcal{I}_j^y$ to $\mathcal{I}_i^x$ in the explored graph.

Proof: ($\Longleftarrow$) Obvious. ($\Longrightarrow$) $\mathcal{I}_j^y \prec \mathcal{I}_i^x$ implies that there is a path from $\mathcal{I}_j^y$ to $\mathcal{I}_i^x$ in the precedence graph. By the definition of explored graph, all vertices on this path together with all edges connecting them should also be in the explored graph. So the same path exists in the explored graph. □

The second difficulty is the potentially expensive space and time cost. The constructed graph expands as new nodes and edges are gathered and accumulated, demanding a space size probably proportional to the execution time of the monitored program. Also, path computation time becomes longer as graph expands. Therefore, without an effective and efficient way to prune the graph, only short-execution-time program can be debugged without suffering from performance degradation.

Note that we cannot discard $\mathcal{I}_i^x$ and associated incident edges just because e_i^x is found to be removable (or, non-removable), since $\mathcal{I}_i^x$ may be a node on some other cycles. We prune the constructed graph $G_c = (\mathcal{I}_c, E_c)$ by removing any subgraph $G_e = (\mathcal{I}_e, E_e)$ that is an explored graph from it. The remaining graph is the induced subgraph of G_c on the vertex set $\mathcal{I}_c - \mathcal{I}_e$. This pruning scheme is repeatedly performed to prevent the constructed graph from growing unlimitedly. After a subgraph has been identified as being an explored graph but before it is removed, we examine every event interval $\mathcal{I}_i^x$ in it to see if there exists a path from $\mathcal{I}_i^x$ to $\mathcal{I}_i^{x-1}$. We claim that our graph-pruning scheme is safe. That is, it neither causes non-useless events to be wrongly identified as useless, nor does it hide any useless events from being detected. Detailed proof is given in Section 3.4.

3. The proposed algorithm

Our algorithm consists of two modules. The first module, cooperatively executed by non-checker and checker processes, collects direct precedence relation information and forms the constructed graph. The second module identifies useless predicate events by examining the constructed graph, and prunes the graph by removing explored subgraph that has been completely examined. These two modules are not assumed to be executed in parallel: only one module can be activated at any instant of time. We do not prescribe any particular scheduling policy for them.

3.1. Collecting direct precedence information

Each non-checker process locally tracks direct precedence relation resulted from the execution of the monitored program. The checker process collects pieces of direct precedence relation from all non-checker processes and assemble them into a constructed graph. The operation performed by each non-checker process is as follows (refer to Figure 2). P_i uses integer variable C_i to count the number of predicate events detected at P_i so far. An event interval (EI) tuple $\langle i, C_i \rangle$, which uniquely identifies P_i's current event interval, $\mathcal{I}_i^{C_i}$, is attached to every application message sent by P_i. P_i uses set variable $\mathcal{R}_i$ to hold all incoming EI tuples. The contents of $\mathcal{R}_i$ thus correspond to event intervals immediately preceding $\mathcal{I}_i^{C_i}$—EI tuple $\langle j, y \rangle$ is in $\mathcal{R}_i$ only if event interval $\mathcal{I}_j^y$ immediately precedes $\mathcal{I}_i^{C_i}$. P_i concludes

Init
    $\mathcal{R}_i \leftarrow \{\}$
    $C_i \leftarrow 0$

On sending a message m
    attach $\langle i, C_i \rangle$ to m

On receiving an EI tuple $\langle j, y \rangle$
    $\mathcal{R}_i \leftarrow \mathcal{R}_i \cup \{\langle j, y \rangle\}$

On detecting a local predicate
    send $\{\mathcal{R}_i, i, C_i\}$ to the checker process as a debug message
    $\mathcal{R}_i \leftarrow \{\langle i, C_i \rangle\}$
    $C_i \leftarrow C_i + 1$

Figure 2. Operations performed by non-checker process P_i

the current event interval upon detecting a local predicate, at which time it sends $\{\mathcal{R}_i, i, C_i\}$ as a debug message to the checker process. After that, $\mathcal{R}_i$ is reset to $\{\langle i, C_i \rangle\}$, indicating that $\mathcal{I}_i^{C_i}$ is the only event interval currently known to immediately precede the next one, and C_i is then increased by one.

3.2. Assembling constructed graph

Upon receiving a debug message, the checker process invokes procedure *update_graph*, as shown in Figure 3, to update the constructed graph. The resulted graph is represented by a matrix, A. We denote the row of A associated with $\mathcal{I}_i^x$ by $A[\mathcal{I}_i^x, *]$ and the column associated with $\mathcal{I}_i^x$ by $A[*, \mathcal{I}_i^x]$. The rows and columns of A are dynamically added and deleted. Initially, A is a null matrix with no columns and rows. Set variables *col_items* and *row_items*, initially empty, are used to hold the sets of event intervals currently contained among the columns and rows, respectively.

Procedure *update_graph* operates as follows. Let the received message be $\{\mathcal{R}_i, i, x\}$. The procedure first adds a new column, $A[*, \mathcal{I}_i^x]$, to matrix A. It then examines each EI tuple $\langle j, y \rangle$ in $\mathcal{R}_i$ to see if it is necessary to add row $A[\mathcal{I}_j^y, *]$ to A as well. The addition is not necessary if $A[\mathcal{I}_j^y, *]$ already exists or once existed but has been discarded (by our graph-pruning scheme, discussed later). Integer array *max_discarded* is used to prevent discarded row items from being added again. *max_discarded*$[j]$ records the maximal t such that event intervals $\mathcal{I}_j^0, \mathcal{I}_j^1, \ldots, \mathcal{I}_j^t$ all have been discarded. Our graph-pruning scheme ensures that an event interval will not be discarded if any event interval preceding it has not yet been discarded. So $y > max\_discarded[j]$ means that $A[\mathcal{I}_j^y, *]$ has not ever been discarded.

It is not hard to see that *col_items* corresponds to the vertex set C of the constructed graph. Also, the set corresponding to the vertex set R is *row_items* $-$ (*col_items* $\cap$ *row_items*).

Init
    max_discarded$[j] = -1$ for all j
    row_items $\leftarrow \{\}$
    col_items $\leftarrow \{\}$

On receiving $\{\mathcal{R}_i, i, x\}$
    add a new column $A[*, \mathcal{I}_i^x]$ to A
    col_items $\leftarrow$ *col_items* $\cup \{\mathcal{I}_i^x\}$
    for each EI tuple $\langle j, y \rangle \in \mathcal{R}_i$ **do**
        if $y >$ *max_discarded*$[j]$ **then**
            if $\mathcal{I}_j^y \notin$ *row_items* **then**
                add a new row $A[\mathcal{I}_j^y, *]$ to A
                row_items $\leftarrow$ *row_items* $\cup \{\mathcal{I}_j^y\}$
            end if
            $A[\mathcal{I}_j^y, \mathcal{I}_i^x] \leftarrow 1$
        end if
    end for

Figure 3. Procedure *update_graph*

3.3. Finding useless events and pruning the graph

The checker process invokes procedure *find_removable* (Figure 4) to perform two tasks: to find and remove useless predicate events pending in event queues and to identify and remove explored subgraph from the constructed graph. The first task is done by first computing the transitive closure of A and then checking whether $A[\mathcal{I}_i^{x+1}, \mathcal{I}_i^x] = 1$ for every $\mathcal{I}_i^x$ in *col_items*. This approach would take $\Theta(|col\_items|^2 |row\_items|)$ computation time. Alternatively, we can determine whether the predicate event corresponding to $\mathcal{I}_i^x$ is removable by searching a path from $\mathcal{I}_i^{x+1}$ to $\mathcal{I}_i^x$. We may set $A[\mathcal{I}', \mathcal{I}]$ to 1 when a path from $\mathcal{I}'$ to $\mathcal{I}$ is found during the search. This action gathers partial transitivity information as the search progresses and thus facilitates the searching.

While checking whether the predicate event corresponding to $\mathcal{I}_i^x$ is removable, we also check whether $\mathcal{I}_i^x$ is a part of an explored graph. This can be done by checking whether all $\mathcal{I}_j^y$ in *row_items* that precede $\mathcal{I}_i^x$ are also in *col_items* (function *is_explored* in Figure 4). Such $\mathcal{I}_i^x$ is called an explored event interval. All explored event intervals found are stored in set variable *explored*, and will be deleted one by one from the constructed graph.

3.4. Proof of safety

In this section, we shall prove that our graph-pruning scheme neither causes non-useless events to be wrongly identified as useless, nor does it hide any useless events from being detected.

```
procedure find_removable
  /* compute the transitive closure of A */
    for each I ∈ col_items do
        for each I' ∈ row_items do
            if A[I', I] = 1 and I ∈ row_items then
                for each I'' ∈ col_items do
                    if A[I, I''] = 1 then A[I', I''] ← 1
  /* identify useless events and explored event intervals */
    explored← {}
    for each I_i^x ∈ col_items do
        if is_explored(I_i^x) then explored← explored ∪{I_i^x}
        if I_i^{x+1} ∈ row_items and A[I_i^{x+1}, I_i^x] = 1 then
            remove e_i^{x+1} from Q_i if e_i^{x+1} exists
    end for
  /* discard explored event intervals */
    for each I_i^x ∈explored do
        remove A[I_i^x, *] and A[*, I_i^x] from A
        row_items ← row_items −{I_i^x}
        col_items ← col_items −{I_i^x}
        if x >max_discarded[i] then max_discarded[i] ← x
    end for
end.

function is_explored(I): Boolean
    for each I' ∈ row_items such that A[I', I] = 1 do
        if I' ∉ col_items then return false
    end for
    return true
end.
```

Figure 4. Procedure *find_removable*

First observe that property $A[\mathcal{I}_j^y, \mathcal{I}_i^x] = 1 \Longrightarrow \mathcal{I}_j^y \prec \mathcal{I}_i^x$ holds before any event interval has ever been discarded. Since each entry $A[\mathcal{I}_j^y, \mathcal{I}_i^x]$ is either discarded or left unchanged but not modified, this property still holds after the first event interval is discarded. It also holds after successive removal of event intervals for the same reason. This implies that no predicate event will be wrongly identified as useless due to discarding explored event intervals.

To prove that the graph-pruning scheme does not hide any useless events from being detected, we show that if e_i^{x+1} is useless, then $A[\mathcal{I}_i^{x+1}, \mathcal{I}_i^x] = 1$ when $\mathcal{I}_i^x$ is identified as explored. Let E^n denote the set of event intervals identified as explored in the n-th invocation of procedure *find_removable*. Let $D^n = E^1 \cup E^2 \cup \cdots \cup E^n$ denote the set of event intervals that have ever been discarded from the very beginning up to the end of the n-th invocation of procedure *find_removable*. The proof is divided into two steps. First we show that if e_i^{x+1} is useless, $\mathcal{I}_i^x \in E^n$ implies that $\mathcal{I}_i^{x+1} \notin D^{n-1}$. Then we show that if $\mathcal{I}_i^x \in E^n$, for any other event interval $\mathcal{I}_j^y \notin D^{n-1}$, $\mathcal{I}_j^y \prec \mathcal{I}_i^x$ implies that $A[\mathcal{I}_j^y, \mathcal{I}_i^x] = 1$ when $\mathcal{I}_i^x$ is identified as explored.

The following statements lead to the first step of the proof. If we do not discard any explored event intervals, the sufficient condition for identifying that $\mathcal{I}$ is explored, i.e., $\mathcal{I}'$ is in *col_items* for all $\mathcal{I}'$ such that $A[\mathcal{I}', \mathcal{I}] = 1$, essentially implies that all event intervals preceding $\mathcal{I}$ are in *col_items*. This implication must be revised when we do discard explored event intervals. In such cases, some event intervals preceding $\mathcal{I}$ may be discarded before $\mathcal{I}$ is identified as explored. Therefore, the same sufficient condition for identifying that $\mathcal{I}$ is explored now implies that all event intervals preceding $\mathcal{I}$ either are in *col_items* or have been discarded. The behavior of function *is_explored* can be described as follows.

Theorem 2 $\mathcal{I} \in E^n \Longrightarrow \forall \mathcal{I}', \mathcal{I}' \prec \mathcal{I} : \mathcal{I}' \in$ *col_items* $\vee \mathcal{I}' \in D^{n-1}$

Theorem 3 $\forall \mathcal{I}, \mathcal{I}' : \mathcal{I}' \prec \mathcal{I} :: \mathcal{I} \in E^n \Longrightarrow \mathcal{I}' \in E^m$, where $m \leq n$.

Proof: For any event interval $\mathcal{I}'$ that precedes $\mathcal{I}$, we have $\{\mathcal{I}'\} \cup \{\mathcal{I}'' | \mathcal{I}'' \prec \mathcal{I}'\} \subseteq \{\mathcal{I}'' | \mathcal{I}'' \prec \mathcal{I}\}$. By Theorem 2, $\mathcal{I} \in E^n$ only if $\{\mathcal{I}'' | \mathcal{I}'' \prec \mathcal{I}\} \subseteq$ *col_items* $\cup D^{n-1}$ in the n-th invocation. This necessity implies that $\{\mathcal{I}'\} \cup \{\mathcal{I}'' | \mathcal{I}'' \prec \mathcal{I}'\} \subseteq$ *col_items* $\cup D^{n-1}$. Consequently, either $\mathcal{I}'$ will also be identified as explored and discarded in the same invocation, or $\mathcal{I}'$ has already been discarded. Thus we have $\mathcal{I}' \in E^m$, where $m \leq n$. □

Corollary 1 $\forall \mathcal{I}, \mathcal{I}' : \mathcal{I} \prec \mathcal{I}' \wedge \mathcal{I}' \prec \mathcal{I} :: \mathcal{I} \in E^n \Longleftrightarrow \mathcal{I}' \in E^n$.

Corollary 2 $\forall \mathcal{I}, \mathcal{I}' : \mathcal{I} \prec \mathcal{I}' \wedge \mathcal{I}' \prec \mathcal{I} :: \mathcal{I} \in D^n \Longleftrightarrow \mathcal{I}' \in D^n$.

Corollary 1 indicates that if e_i^{x+1} is useless, $\mathcal{I}_i^{x+1}$ has not yet been discarded when $\mathcal{I}_i^x$ is identified as explored. This completes the first-step proof.

Next we present the second-step proof. All following results hold for the execution period after the transitive closure of A has been computed, but before any explored event interval found is discarded. We define a sequence of event intervals $\{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_n\}$ to be an *event interval chain* if $\mathcal{I}_1 \prec_d \mathcal{I}_2, \mathcal{I}_2 \prec_d \mathcal{I}_3, \ldots,$ and $\mathcal{I}_{n-1} \prec_d \mathcal{I}_n$.

Lemma 2 For any event interval chain $\{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_m\}$, if $\mathcal{I}_1$ has not yet been discarded and $\{\mathcal{I}_2, \mathcal{I}_3, \ldots, \mathcal{I}_m\} \subseteq$ *col_items*, then $A[\mathcal{I}_1, \mathcal{I}_m] = 1$ after the transitive closure of A has been computed.

Proof: The proof is by induction on m. The claim trivially holds for $m = 2$. We assume that the claim holds for $m < k$ and consider the case of $m = k$. Let $\{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_k\}$ be an event interval chain such that $\mathcal{I}_1$ has not yet been discarded and $\{\mathcal{I}_2, \mathcal{I}_3, \ldots, \mathcal{I}_k\} \subseteq$ *col_items*. Let $\mathcal{I}_i \in \{\mathcal{I}_2, \mathcal{I}_3, \ldots, \mathcal{I}_k\}$ be the last event interval that was added to *col_items*. Just before the addition, we have

$\{\mathcal{I}_2, \mathcal{I}_3, \ldots, \mathcal{I}_{i-1}\} \subseteq$ *col_items*. By the induction hypothesis, this implies that $A[\mathcal{I}_1, \mathcal{I}_{i-1}] = 1$ at that time. Similarly, we have $\{\mathcal{I}_{i+1}, \mathcal{I}_{i+2}, \ldots, \mathcal{I}_k\} \subseteq$ *col_items*. By Corollary 2, $\mathcal{I}_{i+1}$ had not been discarded at that time, since $\mathcal{I}_i$, the event interval immediately preceding $\mathcal{I}_{i+1}$, is not even received. By the induction hypothesis, this implies that $A[\mathcal{I}_{i+1}, \mathcal{I}_k] = 1$ at that time. After the edge $(\mathcal{I}_{i-1}, \mathcal{I}_i)$ has been added to the graph, having computed the transitive closure of A ensures that $A[\mathcal{I}', \mathcal{I}] = 1$ for all $\mathcal{I}'$ such that $A[\mathcal{I}', \mathcal{I}_{i-1}] = 1$ and for all $\mathcal{I}$ such that $A[\mathcal{I}_{i+1}, \mathcal{I}] = 1$. Thus $A[\mathcal{I}_1, \mathcal{I}_k] = 1$ after the addition. It remains to be so as long as $\mathcal{I}_1$ is not discarded and $\mathcal{I}_k \in$ *col_items*. □

Theorem 4 For every $\mathcal{I} \in$ *col_items* and every $\mathcal{I}'$ that has not been discarded, if $\mathcal{I}' \prec \mathcal{I}$ and $A[\mathcal{I}', \mathcal{I}] \neq 1$, then there must be some $\mathcal{I}''$ such that $A[\mathcal{I}'', \mathcal{I}] = 1$ and $\mathcal{I}''$ is not in *col_items*.

Proof: $\mathcal{I}' \prec \mathcal{I}$ implies that there exists at least one event interval chain $\{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_m\}$ $(m \geq 2)$ such that $\mathcal{I}' \equiv \mathcal{I}_1$ and $\mathcal{I}_m \equiv \mathcal{I}$. Consider any such chain. Since $\mathcal{I}'$ has not been discarded, by Lemma 2, $A[\mathcal{I}', \mathcal{I}] \neq 1$ implies that there must be some event interval in $\{\mathcal{I}_2, \mathcal{I}_3, \ldots, \mathcal{I}_{m-1}\}$ that is not in *col_items*. Let s be the maximal index of such event intervals. If we can show that $A[\mathcal{I}_s, \mathcal{I}_m] = 1$, the proof is done. Observe that $\mathcal{I}_s$ has not been discarded since $\mathcal{I}_s$ is not even received. Also, $\{\mathcal{I}_{s+1}, \mathcal{I}_{s+2}, \ldots, \mathcal{I}_m\} \subseteq$ *col_items*. By Lemma 2, we then have $A[\mathcal{I}_s, \mathcal{I}_m] = 1$. □

Corollary 3 For every $\mathcal{I} \in$ *col_items*, if there exists no event interval $\mathcal{I}''$ such that $A[\mathcal{I}'', \mathcal{I}] = 1$ and $\mathcal{I}'' \notin$ *col_items*, then there exists no $\mathcal{I}'$ such that $\mathcal{I}'$ has not been discarded and $\mathcal{I}' \prec \mathcal{I} \wedge A[\mathcal{I}', \mathcal{I}] \neq 1$.

Theorem 5 For every $\mathcal{I} \in$ *col_items*, if *is_explored*($\mathcal{I}$) returns *true*, then for every $\mathcal{I}$ that has not been discarded, we have $\mathcal{I}' \prec \mathcal{I} \Longrightarrow A[\mathcal{I}', \mathcal{I}] = 1$.

Proof: For every event interval $\mathcal{I} \in$ *col_items*, if *is_explored*($\mathcal{I}$) returns true, then there exists no $\mathcal{I}''$ such that $A[\mathcal{I}'', \mathcal{I}] = 1$ and $\mathcal{I}'' \notin$ *col_items*. By Corollary 3, this implies that there exists no $\mathcal{I}'$ such that $\mathcal{I}'$ has not been discarded and $\mathcal{I}' \prec \mathcal{I} \wedge A[\mathcal{I}', \mathcal{I}] \neq 1$, which in turn implies that for every $\mathcal{I}'$ that has not been discarded, $\mathcal{I}' \prec \mathcal{I} \Longrightarrow A[\mathcal{I}', \mathcal{I}] = 1$. □

By Corollary 1 and Theorem 5, it is clear that if e_i^{x+1} is useless, then $A[\mathcal{I}_i^{x+1}, \mathcal{I}_i^x] = 1$ when $\mathcal{I}_i^x$ is identified as explored. So e_i^{x+1} will be detected before discarding $\mathcal{I}_i^x$.

4. Performance evaluation

4.1. Time complexity analysis

Assuming that the time to add a column or row is $O(1)$ (which can be achieved if adding a column or row is in fact adding a head of a linked list), the computation time of procedure *update_graph* is $O(|\mathcal{R}_i|)$ for a single debug message $\{\mathcal{R}_i, i, x\}$. For a constructed graph $G = (V, E)$, the total computation time for procedure *update_graph* is therefore $O(|E|)$.

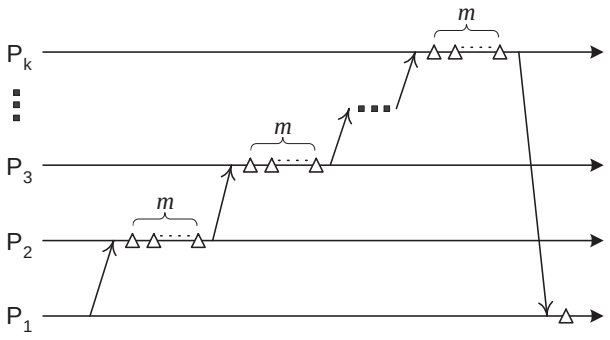

Figure 5. Scenario corresponding to worst-case explored graph

Procedure *find_removable* contains mainly three independent loops. The first loop, which computes the transitive closure of A, takes $\Theta(|\textit{col\_items}|^2|\textit{row\_items}|)$ computation time. If the computation time of checking whether an element is in a given set is proportional to the size of the set, function *is_explored* takes $O(|\textit{col\_items}||\textit{row\_items}|)$ time. So the second loop, which identifies explored event intervals and useless predicate events, takes $O(|\textit{col\_items}|^2|\textit{row\_items}|)$ computation time. The third loop obviously takes less computing time. We have that *find_removable_A* is of $\Theta(|\textit{col\_items}|^2|\textit{row\_items}|)$ time complexity.

The values of $|$*col_items*$|$ and $|$*row_items*$|$ depend on whether the constructed graph contains many small subgraphs that are explored graphs. If it does, every time we execute procedure *find_removable*, we can hopefully remove some explored subgraphs from it and thus shrink sets *col_items* and *row_items*. On the other hand, if it does not, our pruning scheme will not help in decreasing $|$*col_items*$|$ and $|$*row_items*$|$.

Suppose that k event queues are non-empty, each of which has m events. The worst case we may encounter is that all events in $k-1$ out of the k event queues form an explored graph that cannot be divided into smaller explored subgraphs (see Figure 5). In this case, the maximal possible values of $|$*col_items*$|$ and $|$*row_items*$|$ are both $(k-1)m$, resulting in $O(k^3m^3)$ time complexity for *find_removable*.

The result obtained is for a single invocation of procedure *find_removable*. The overall execution time depends on how frequently we execute the procedure. If we execute it too often, the benefit we obtain from cutting off explored

subgraph may not compensate the execution time imposed on. On the other hand, if we seldom invoke the procedure, the execution time may be terrible when we indeed execute it. The optimal solution cannot be obtained if we do not know in advance what the constructed graph will be.

4.2. Simulation results

In this section, we evaluate the performance of our approach by simulation. We assume a system consisting of ten processes. All processes but one randomly and independently generate predicate events, and report event intervals to the checker process in the order as they occur. The time interval between two consecutive predicate events is assumed to be an exponentially distributed random variable with mean α_1. Each process randomly and independently sends messages, with destinations uniformly distributed over all processes other than the sender. Message propagation delays and the time interval between two consecutive sending events are both assumed to be exponentially distributed, with parameters α_3 and α_2, respectively.

We varied the values of α_1, α_2, and α_3 to represent various types of computation and communication patterns. We computed the ratio of predicate events identified useless to the total. This is used as the metrics to compare our approach with the two methods proposed by Chiou and Korfhage, which are respectively denoted by C&K(A) and C&K(B). The difference between C&K(A) and C&K(B) is that the latter performs the same check repeatedly, rather than just once, until no further useless events can be found. Figures 6-9 show some representative results.

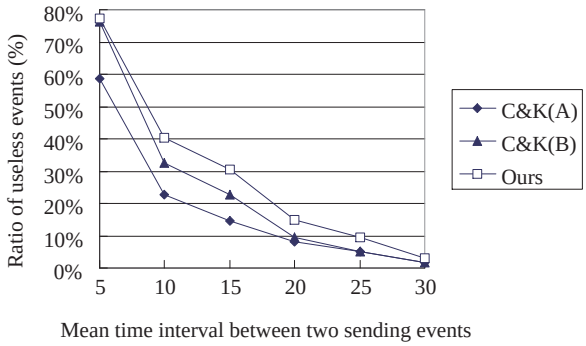

Figure 6. Results with $\alpha_1 = 15$ and $\alpha_3 = 5$

Clearly, with a fixed α_3, the ratio of useless predicate events decreases as the ratio α_1/α_2 increases (Figures 6 and 7). This can be explained as follows. When $\alpha_1/\alpha_2 < 1$, many processes do not send out any application message between two consecutive predicate events, so zigzag cycles

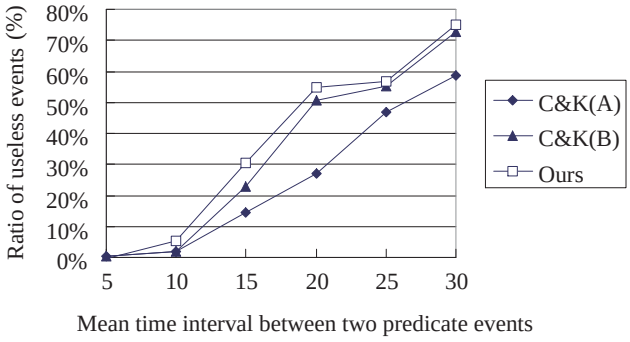

Figure 7. Results with $\alpha_2 = 15$ and $\alpha_3 = 5$

are unlikely to happen, and few useless events can be found. On the other hand, when $\alpha_1/\alpha_2 > 1$, messages are likely to be sent between two consecutive predicate events, implying that zigzag cycles and thus useless events are more possible to occur. Mean message propagation delays (α_3) also have impacts on the ratio of useless events. Generally, with a fixed α_1/α_2 setting, the ratio of useless predicate events decreases as α_3 increases (Figures 8 and 9).

In all settings, our approach outperforms C&K(A) in terms of the number of useless events found. The performance of C&K(B) is between ours and C&K(A). When the ratio of useless events is high, C&K(B) is a match for our approach. But it degrades to C&K(A) when the ratio becomes low.

We also found that our graph-pruning technique does not help much in shrinking the size of A. The reason is that we let one of all processes send messages but not report any predicate event. As a result, there did exist a large subgraph that cannot be divided into smaller explored subgraphs (i.e., the one shown in Figure 5). That graph cannot be constructed by the checker process and found to be explored, since there is always one event interval absent.

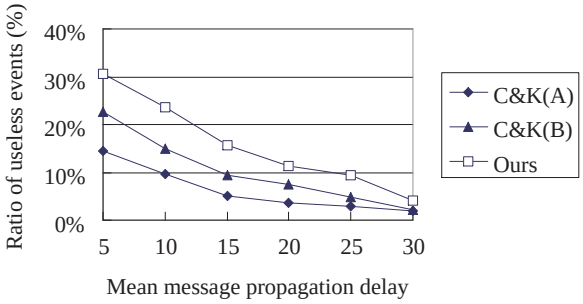

Figure 8. Results with $\alpha_1 = 15$ and $\alpha_2 = 15$

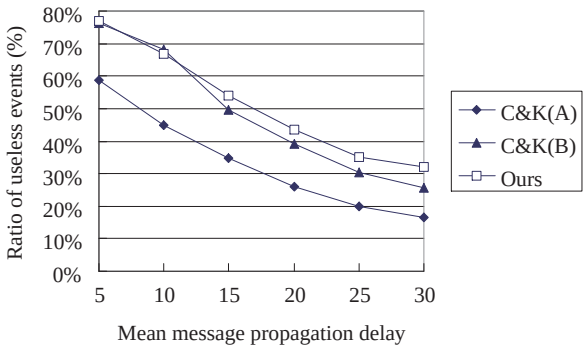

Figure 9. Results with $\alpha_1 = 15$ and $\alpha_2 = 5$

5. Conclusions

We have proposed an approach that effectively precludes useless events for global predicate detection, facilitating the process of an independent on-line checking routine. To identify more useless events than a simple causality-check method can do, our method tracks and maintains the precedence information of event intervals as a graph. To reduce the potentially expensive space and time cost as the graph expands, we have proposed a safe scheme to prune the graph. This scheme is safe in the sense that it neither causes non-useless events to be wrongly identified as useless, nor does it hide any useless events from being detected.

Suppose that k event queues are non-empty, each of which has m events. Our approach takes $O(k^3m^3)$ computation time, while Chiou and Korfhage's method takes $O(k^2m^2)$. Our method, however, does not necessarily increase additional computation overhead on the checker process, since the checker process is otherwise idle. The simulation result shows that our approach outperforms conventional approaches in terms of the number of useless events found.

References

[1] R. Baldoni, J.-M. Helary, and M. Raynal. About state recording in asynchronous computations. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, 1996.

[2] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[3] H.-K. Chiou and W. Korfhage. Efficient global event predicate detection. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 642–649, June 1994.

[4] H.-K. Chiou and W. Korfhage. Enhancing distributed event predicate detection algorithms. *IEEE Trans. Parallel Distrib. Syst.*, 7(7):673–676, July 1996.

[5] R. Cooper and K. Marzullo. Consistent detection of global predicates. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, ACM SIGPLAN Notices*, 26(12):167–174, December 1991.

[6] J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, pages 56–66, February 1988.

[7] V. K. Garg. Some optimal algorithms for decomposed partially ordered sets. *Inform. Process. Lett.*, 44(1):39–43, November 1992.

[8] V. K. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Trans. Parallel Distrib. Syst.*, 5(1):299–307, March 1994.

[9] V. K. Garg and B. Waldecker. Detection of strong unstable predicates in distributed programs. *IEEE Trans. Parallel Distrib. Syst.*, 7(12):1323–1333, December 1996.

[10] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):538–565, July 1978.

[11] H. F. Li and B. Dash. Detection of safety violations in distributed systems. In *Proceedings of 1992 International Conference on Parallel and Distributed Systems*, pages 275–282, 1992.

[12] F. Mattern. Virtual time and global states of distributed systems. In M. C. et al., editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, North-Holland, 1989. Elsevier Science.

[13] B. P. Miller and J.-D. Choi. Breakpoints and halting in distributed programs. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 316–323, June 1988.

[14] R. H. B. Netzer and J. Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Trans. Parallel Distrib. Syst.*, 6(2):165–169, February 1995.

[15] S. Venkatesan and B. Dathan. Testing and debugging distributed programs using global predicates. *IEEE Transactions on Software Engineering*, 21(2):163–177, February 1995.