Detection of Strong Unstable Predicates in Distributed Programs

Vijay K. Garg, *Senior Member, IEEE,* and Brian Waldecker, *Member, IEEE Computer Society*

Abstract—This paper discusses detection of global predicates in a distributed program. A run of a distributed program results in a set of sequential traces, one for each process. These traces may be combined to form many global sequences consistent with the single run of the program. A strong global predicate is true in a run if it is true for all global sequences consistent with the run. We present algorithms which detect if the given strong global predicate became true in a run of a distributed program. Our algorithms can be executed on line as well as off line. Moreover, our algorithms do not assume that underlying channels satisfy FIFO ordering.

Index Terms—Unstable predicates, predicate detection, distributed algorithms, distributed debugging.

1 Introduction

Detection of global predicates is a fundamental problem in distributed computing. It arises in the designing, debugging and testing of distributed programs. Global predicates can be classified into two types—stable and unstable. A stable predicate is one which never turns false once it becomes true. An unstable predicate is one without such a property. Its value may alternate between true and false. Detection of stable predicates has been addressed in the literature by means of global snapshots of a distributed computation [5], [22], [4]. Any stable property can be detected by taking global snapshots periodically. This approach does not work for an unstable predicate which may turn true only between two snapshots and not at the time when the snapshot is taken. An entirely different approach is required for such predicates [26], [7], [10], [21], [12], [1], [24], [16].

We have earlier presented an approach to detect a class of unstable predicates called weak predicates [12]. A weak predicate p holds in a distributed computation if there exists a consistent global state in which p is true. This is similar to *possibly* : p as proposed by Cooper and Marzullo [7]. Weak predicates are generally useful in detecting violation of a safety property during a distributed computation. Intuitively, a weak predicate p is true if there exists an observation of the distributed computation in which p is true. In this paper, we continue our investigation of detection for a different class of unstable predicates called strong predicates. Intuitively, a strong predicate p is true if it is true for all observers of the distributed computation. These predicates are generally useful in verifying that a desirable predicate was indeed true in a distributed computation.

Two types of predicates are discussed in this paper. The first type, called strong linked predicates, refers to a causal sequence of local predicates. These predicates were first discussed in [20]. Our contribution is two-fold. First, our algorithm which is a simple variation of Miller and Choi's algorithm has the advantage that it does not assume FIFO channels in the system. Secondly and more importantly, we provide a proof of the correctness of the algorithm. The second type, called strong conjunctive predicates, correspond to existence of a global state in which all local predicates are true simultaneously. We introduce the notion of overlapping intervals which is used to detect predicates of this type. Cooper and Marzullo [7] also describe strong predicate detection (they call such predicates *definitely*). However, they deal with general predicates, i.e., they propose detection of *definitely* : p where p is any predicate defined on a global state. In this paper, we have restricted p to conjunction of local predicates. Detection of general predicates is intractable since it involves a combinatorial explosion of the state space. For example, the algorithm proposed by Cooper and Marzullo [7] has complexity $O(k^n)$ where k is the maximum number of events a monitored process has executed and n is the number of processes. The fundamental difference between our algorithm and their algorithm is that their algorithm explicitly checks all possible global states, whereas our algorithm does not.

Spezialetti and Kearns [23] also discuss a notion of simultaneity which, however, is different from the one discussed in this paper. They use simultaneity in the sense of a possible consistent global state, that is, two states are defined to be simultaneous if no *happened-before* [17] relation can be established between them. Thus, their notion is similar to to *possibly*: p [7] and weak predicates [12]. We restrict ourselves to strong predicates in this paper.

Independent of us, Venkatesan and Dathan [25] consider algorithms for *definitely* : p. However, their algorithm is off line, whereas our algorithm can be executed on line. Further, they assume that all channels in the system satisfy FIFO ordering. We do not make such assumptions. Their algorithm has the advantage that it does not use vector clocks.

The paper is organized as follows: Section 2 presents our logic for specification of global predicates in a distributed

• *V.K. Garg is with the Electrical and Computer Engineering Department, University of Texas, Austin, TX 78712-1084. E-mail: garg@ece.utexas.edu.*
• *B. Waldecker is with International Business Machines, 11400 Burnet Rd., Austin, TX 78758. E-mail: brianw@ibmoto.com.*

Manuscript received July 30, 1993.
For information on obtaining reprints of this article, please send e-mail to: transpds@computer.org, and reference IEEECS Log Number D95214.

program. It describes the notion of a distributed run, a logical clock, a global sequence and the logic. Section 3 discusses detection of linked predicates. Section 4 discusses strong conjunctive predicates. It gives necessary and sufficient conditions for strong conjunctive predicates to hold. It also describes algorithms for detecting strong conjunctive predicates. Section 5 presents techniques to decentralize these algorithms.

2 Our Model

We assume a loosely coupled message-passing system without any shared memory or a logical clock. A distributed program consists of a set of n processes denoted by $\{P_1, P_2, \ldots, P_n\}$ communicating solely via asynchronous messages.

We assume that no messages are lost, altered, or spuriously introduced. We do not make any assumptions about FIFO nature of channels.

To reason about causal sequences of predicates and simultaneous truth of a set of local predicates, we need a model that captures sequencing of events within and between processes. We describe formally a *run* that captures the notion of a single execution of a distributed program, a *logical clock* that captures causal dependencies, a *global sequence* that captures a possible observation of a run, and finally a *logic for global predicates* that captures means of specifying global conditions on a run.

2.1 Run

We will be concerned with a single run r of a distributed program. Each process P_i in that run generates an execution trace $s_{i,0}\, s_{i,1} \ldots s_{i,l}$, which is a finite sequence of local *states* in the process P_i. A *run* r is a vector of traces with $r[i]$ as the trace of the process P_i.

We define a relation *locally precedes* denoted by $\prec_{im}$ between states in the trace of a single process P_i as follows: $s \prec_{im} t$ if and only if s *immediately* precedes t in the trace $r[i]$. We also say that $s.next = t$ or $t.prev = s$ whenever $s \prec_{im} t$. We use $\prec$ for irreflexive transitive closure and $\preceq$ for reflexive transitive closure of $\prec_{im}$. States s and t in the traces $r[i]$ and $r[j]$, respectively, are defined to be related by $\rightsquigarrow$ if and only if a message is sent by P_i resulting in the state s which is received by P_j resulting in the state t. Fig. 1 illustrates a run.

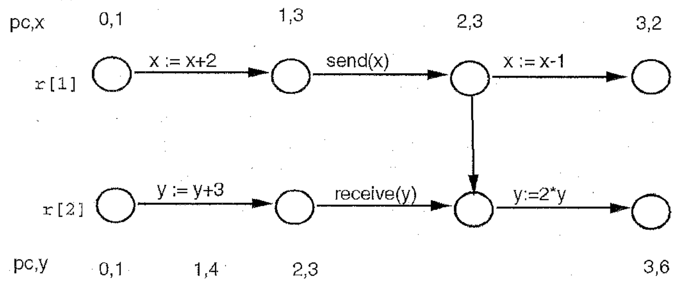


Fig. 1. An example of a run.

We also define a *causally precedes* relation as the transitive closure of union of $\prec_{im}$ and $\rightsquigarrow$. That is,

$s \rightarrow t$ iff

1) $(s \prec_{im} t) \vee (s \rightsquigarrow t)$, or
2) $\exists u : (s \rightarrow u) \wedge (u \rightarrow t)$

Our $\rightarrow$ is similar to Lamport's happened-before relation [17] except that *causally precedes* is defined between states rather than events. We say that s and t are concurrent (denoted by $s \mid\mid t$) if $\neg(s \rightarrow t) \wedge \neg(t \rightarrow s)$.

We extend the run r to r' by adding artificial states $\bot_i$ and $\top_i$ at the beginning and the end of each trace $r[i]$ respectively. The event at $\bot_i(\top_i)$ corresponds to the beginning (termination) of the execution of P_i. The addition of these artificial states model the fact that processes begin their execution asynchronously. Thus, in absence of any synchronization (external events) it may be possible that one process may have terminated while the other one has yet to begin the execution. Let S_i be the set of all states in the sequence $r[i]$, and $S'_i = S_i \cup \{\bot_i, \top_i\}$. Our definitions imply that $\forall s \in S_i : \bot_i \rightarrow s \wedge s \rightarrow \top_i$. We define $S = \bigcup_i S_i$ and $S' = \bigcup_i S'_i$. We also use $s.p$ to denote the process in whose trace s occurs. That is, $s.p = i$ if and only if $s \in S'_i$.

2.2 Logical Clock

A *logical clock* C is a map from S' to $\mathbb{N}$ (the set of natural numbers) with the following constraint:

$$\forall s, t \in S : s \prec_{im} t \vee s \rightsquigarrow t \Rightarrow C(s) < C(t)$$

We use $\mathcal{C}$ to denote the set of all logical clocks which satisfy the above constraint. The interpretation of $C(s)$ for any $s \in S$ is that the process $s.p$ enters the state s when the clock value is $C(s)$. Thus, it stays in the state s from time $C(s)$ to $C(s.next) - 1$. This constraint models the sequential nature of execution at each process and the physical requirement that any message transmission requires a nonzero amount of time. From the definition of $\rightarrow$, it is equivalent to

$$\forall s, t \in S : s \rightarrow t \Rightarrow \forall C \in \mathcal{C} : C(s) < C(t) \qquad \text{(CC)}$$

The condition (CC) is widely used as the definition of a logical clock since its proposal by Lamport [17]. It can be shown that the set $\mathcal{C}$ also satisfies the converse of (CC), i.e.,

$$\forall s, t \in S : s \not\rightarrow t \Rightarrow \exists C \in \mathcal{C} : \neg(C(s) < C(t))$$

The reader is referred to [13] for the proof. This leads to the following pleasant characterization of $\rightarrow$:

$$\forall s, t \in S : (s \rightarrow t \Leftrightarrow \forall C \in \mathcal{C} : C(s) < C(t))$$

Intuitively, the above formula says that s causally precedes t in a run r if and only if all possible observers of the run agree that s happened before t.

2.3 Global Sequence

A global state is a vector of local states. This definition of global state is different from that of Chandy and Lamport which includes states of channels. In our model, a channel is just the set of all those messages that have been sent but not received yet. Since this set can be deduced from all the local states, we do not require the state of channels to be explicitly included in the global state. Given a run r, and a logical clock C, $seq(r, C)$ defines a sequence of global states called global sequence $g = g_0\, g_1 \ldots g_m$ for some m where

$$g_k[i] = max\{\{s \in S_i' | C(s) \le k\} \cup \{\perp_i\}\},\quad 1 \le k \le m,\ 1 \le i \le n$$

Note that $g_k[i]$ is well defined as the argument of max is a nonempty totally ordered (under $\prec$) finite set. It may evaluate to $\perp_i$ which would mean that the process P_i has not begun its execution. Similarly, if it evaluates to $\top_i$, then process P_i has already terminated its execution. The kth prefix of g, i.e., $g_0\, g_1 \ldots g_{k-1}$ is denoted by g^k.

We define $findex(g, u) = min\ \{k \mid g_k[u.p] = u\}$, i.e., the first index in g which has u in its global state. We define the set of global sequences consistent with a run r as $linear(r)$, i.e.,

$$g \in linear(r) \Leftrightarrow \exists\, C \in \mathcal{C}\ : g = seq(r, C)$$

The following theorem gives an alternative characterization of the set $linear(r)$. Given any $g \in linear(r)$ if the observer restricts his attention to a single process P_i, then he would observe $r[i]$ or a stutter of $r[i]$. A *stutter* of $r[i]$ is a finite sequence where each state in $r[i]$ may be repeated a finite number of times.

LEMMA 1. *For any run r, $g \in linear(r)$ if and only if the following constraints hold:*

(S1): $\forall I : g$ *restricted to* $P_i = r[i]$ (*or a stutter of* $r[i]$)
(S2): $\forall s, t \in S : s \rightarrow t \Rightarrow findex(g, s) < findex(g, t)$.

PROOF. Let $g \in linear(r)$. This implies that $\exists C : g = seq(r, C)$. Since $C(s)$ is greater than 0 for all $s \neq \perp_i$, we get that $g_0[i] = \perp_i$. Further, $g_k[i] = s \Rightarrow g_{k+1}[i] \in \{s, s.next\}$ by definition of $seq(r, C)$. Finally, $g_{C[\top_i]}[i] = \top_i$. Thus, (S1) holds. To see (S2), let $s \rightarrow t$. From (CC), we get that $C(s) < C(t)$. This implies that $findex(g, s) < findex(g, t)$.

($\Leftarrow$)

We define C as follows:

$$C(s) = findex(g, s)$$

C satisfies (CC) due to (S2). □

In our earlier paper [12], we have directly defined the notion of global sequences. In this paper, we have chosen the condition (CC) based on logical clocks as it is intuitively easier to justify.

From the above two properties of global sequences, we can also deduce **(S3)**:

$$\forall X \subseteq S : (\forall u, v \in X : u \,\|\, v) \Rightarrow \exists g \in linear(r)\ \exists k \forall u \in X : (g_k[u.p] = u) \wedge (g_{k-1}[u.p] \neq u).$$

(S3) says that for any set X of concurrent states there exists a global sequence g which goes through a global state g_k such that all local states in X occur in g_k, and none occur in g_{k-1}.

2.4 Logic for Global Predicates

We now describe our logic for specification of global predicates. There are three syntactic categories in our logic: *bool*, *lin*, and *form*. The syntax of our logic is as follows:

form ::= $\underline{A}$: lin | $\underline{E}$: lin
lin ::= (lin | lin $\hookrightarrow$ lin | lin $\wedge$ lin | $\neg$lin | bool
bool ::= a predicate over a global state

A *bool* is a Boolean expression defined on a single global state of the system. Its value can be determined if the global state is known. For example, if the global state has ($x = 3$, $y = 6$), then the *bool* ($x \le y$) is true. Here x and y could be part of different processes. A *lin* is a temporal formula defined over a global sequence. A *bool* is true in a global sequence if it is true in the last state of g. $\Diamond$ *lin* means that there exists a prefix of the global sequence such that *lin* is true for the prefix. We also use $\Box$ and $\vee$ as duals of $\Diamond$ and $\wedge$. We have introduced a binary operator ($\hookrightarrow$) to capture sequencing directly. $p \hookrightarrow q$ means that there exist prefixes g^i and g^j of the global sequence such that p is true of prefix g^i, q is true of prefix g^j, and $i < j$.

A *form* is defined for a run and it is simply a *lin* qualified $\underline{A}$, and $\underline{E}$ quantify over the set of global sequences that a run may exhibit, given the traces for each process. $\underline{A} : p$ means that predicate p holds for all global sequences and $\underline{E} : p$ means that predicate p holds for some global sequence. We call formulas starting with $\underline{A}$: as *strong* formulas and formulas starting with $\underline{E}$: as *weak* formulas. The intuition behind the term *strong* is that a strong predicate is true no matter how fast or slow the individual processes in the system execute so long as the execution is consistent with the run. That is, it holds for all execution speeds which generate the same trace for an individual process. A *weak* predicate is true if and only if there exists at least one global sequence in which it is true. In other words, the predicate can be made true by choosing appropriate execution speeds of various processors.

Semantics defined in this paper are slightly different from that in [12]. In [12], *bool* is defined to be true on a global sequence if it is true in the first global state in the sequence. In this paper, *bool* is required to be true in the last global state. The current version is more useful and easier to understand. Intuitively, the logic in [12] is based on future while the logic in this paper is based on past. Since the past is known at any point of execution, it is easier to evaluate the formula in the current state.

Following are some examples of the strong formulas detectable by our algorithms.

1) Suppose we have developed an algorithm which works in phases. Assume that the system has three nodes and that there are three phases in the algorithm. Let predicate $phase_{i,j}$ denote that the process P_i is in phase j. The following formula ensures that the process P_2 is in phase 3, only after all the processes have been through phase 2.

$$(\underline{A} : phase_{1,2} \hookrightarrow phase_{2,3}) \wedge (\underline{A} : phase_{2,2} \hookrightarrow phase_{2,3}) \wedge (\underline{A} : phase_{3,2} \hookrightarrow phase_{2,3})$$

2) Suppose we were testing a commit protocol. Let $Ready_i$ denote the local predicate that the process P_i is ready to commit. Then, the following formula would check that there was a certain point in the execution when all processes were ready to commit.

$$\underline{A} : \Diamond\, (Ready_1 \wedge Ready_2 \ldots \wedge Ready_n)$$

3) Suppose we wanted to test a distributed minimum spanning tree algorithm. Let K_i represent the local predicate that the process P_i knows its parent. Then, the following formula would indicate that the system has reached a state in which all nodes in the network know their parents.

$$\underline{A}: \Diamond (K_1 \wedge K_2 \wedge \ldots \wedge K_n).$$

3 Linked Predicates

This class of predicates is useful in detecting a sequence of events in a distributed program. We use LP_i to denote a local predicate in some process, and $LP_i(s)$ to denote that the predicate LP_i is true in the state s. We assume that the local predicate LP_i is constructed from only the local variables of that process. This means that the truthfulness of LP_i can change only through an internal event. In other words, external events cannot make any local predicate change from true to false or vice-versa. Thus, a predicate such as "a message has been sent from P to Q" is not considered a valid LP. Although this appears to be a limitation, the above predicate can be easily modeled in our framework by assuming that an internal event records the send of the message in some Boolean variable such as *msg_sent*. The condition *msg_sent* is a valid local predicate. The above assumption is equivalent to the following:

(A1) If $(LP(s) \wedge \neg LP(s.next)) \vee (\neg LP(s) \wedge LP(s.next))$ then

1) $t \Rightarrow s$ iff $t \rightarrow s.next$, and
2) $s \rightarrow t$ iff $s.next \rightarrow t$ for all t different from s and $s.next$.

(A1) says that if s and $s.next$ differ in their evaluation of LP, then their causal relationships with other states is identical.

We also use the following assumption.

(A2) All LPs evaluate to false in the artificial states $\bot_i$ and $\top_i$ for all i. This assumption is also not a restriction. It just captures the intent of defining $\bot_i$ and $\top_i$ states.

A predicate of the form $\underline{A} : LP_i \hookrightarrow LP_j$ means that for all global sequences, there exists an instance where LP_i is true before LP_j. $\underline{A} : (LP_i \hookrightarrow LP_j) \hookrightarrow LP_k$ means that for all global sequences there exists an instance where LP_i is true before LP_j which is true before LP_k. We treat $\hookrightarrow$ as a left associative operator and leave out the parentheses. We call a formula of the form $\underline{A} : LP_1 \hookrightarrow LP_2 \hookrightarrow \cdots \hookrightarrow LP_m$ a *strong* linked predicate. The following theorem is used in designing the algorithm for the detection of such predicates. Note that one side of the proof ($\Leftarrow$) is obvious. The converse, which is more difficult, has not been addressed in the literature. This is one of the main results of this section.

THEOREM 2. *Let LP_1 and LP_2 be local predicates on processes i and j, respectively. Then, for any run r, there exist states s_i in $r[i]$ and s_j in $r[j]$, such that $s_i \rightarrow s_j$, $LP_1(s_i)$ and $LP_2(s_j)$ if and only if $\underline{A} : LP_1 \hookrightarrow LP_2$.*

PROOF. ($\Rightarrow$) Since $s_i \in P_i$ and $s_j \in P_j$, from (S1) we conclude that any global sequence $g \in linear(r)$ has states g_k and g_l such that $g_k[i] = s_i$ and $g_l[j] = s_j$. From (S2), we know that $findex(g, s_i) < findex(g, s_j)$. Thus, $g \models LP_1 \hookrightarrow LP_2$ is true.

($\Leftarrow$) We show that if such states do not exist, then the formula $\underline{A} : LP_1 \hookrightarrow LP_2$ is false (that is $\neg LHS \Rightarrow \neg RHS$). If LP_1 (or LP_2) is not true for any state in $r[i]$ ($[r[j]$, respectively), then the formula is trivially false. Consider the first state in $r[i]$ in which LP_1 is true. We call this state s_i. Similarly, s_j is defined using the last state in $r[j]$ in which LP_2 is true. The negation of left hand side implies that $s_i \not\rightarrow s_j$. Consider the state $s_j.next$. This state exists by (A2); it may be $\top_j$. Let t be defined as

$$t = \min_{s \in S_i} \{s \| s_j.next\}$$

Note that t could possibly be $\bot_i$. See Fig. 2.

We now do case analysis.

Case 1: $s_i \prec t$.

This means that s_i is not concurrent with $s_j.next$ by the definition of t. Since $s_i \not\rightarrow s_j$, we get that $s_i \not\rightarrow s_j.next$ from (A1). This implies that $s_j.next \rightarrow s_i$ which in turn implies that $s_j.next \rightarrow t$, a contradiction.

Case 2: $t \preccurlyeq s_i$

Since t is concurrent with $s_j.next$, by Lemma (S3), there exists a global sequence g in which t and $s_j.next$ occur in the same global state for the first time. The predicate LP_1 is not true for all preceding global states and LP_2 is false for all following global states. Thus, there are no two global states x, y such that $LP_1(x)$, $LP_2(y)$ and x occurs before y in g. □

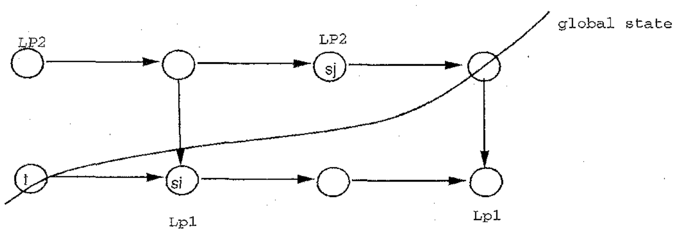


Fig. 2. Linked predicates.

The above result can be generalized to a sequence of more than two local predicates [13].

The intuition behind the algorithm to detect the strong linked predicate (in Fig. 3) is as follows. $\underline{A} : LP_1 \hookrightarrow LP_2$ is true only if the state in which LP_1 has occurred happened before ($\rightarrow$) the state in which LP_2 occurs. If both predicates are in the same process, then occurrence of LP_1 would be known when LP_2 occurs. If LP_2 is in the different process then by the definition of $\rightarrow$ we know that there must be a message path to the second process. We use the same message path to inform the second process about occurrence of LP_1.

The implementation of the algorithm is as follows. The variable *pred_list* in each process keeps the list of logical predicates local to that process in the increasing order of indices in which they appear in strong linked predicate. The variable *curpred* keeps the index of the next local predicate in the strong linked predicate which needs to be detected (as currently known by the process). If curpred becomes $m + 1$ in any process, then the strong linked predicate is detected.

```
P_i::
  var
    detectflag : Boolean always (true iff
      curpred = m + 1);
    pred_list: list of {index:1..m;
      pred : local predicate}
        /* predicates local to this
          process;*/
    curpred: integer initially 1;
□   Upon (head(pred_list).index = curpred)
    ∧ (head(pred_list).pred = true)
    begin /* update what predicate is the
      next one this process is to detect*/
        curpred++;
        pred_list := tail (pred_list);
          /* remove the head */
    end;
□   Upon rcv (prog, hiscurpred, ...) from P_s
      curpred:=max(curpred, hiscurpred);
□   To send /* we include curpred in
      message */
        send(prog, curpred, ...) to destin;
```

Fig. 3. Algorithm for strong linked predicates.

We now show the correctness of the above algorithm. Let *link*(s, j) = $\exists s_1, s_2, ..., s_{j-1} : (s_1 \to s_2) \wedge (s_2 \to s_3) \wedge ... \wedge (s_{j-1} \to s) \wedge LP_1(s_1) \wedge ... \wedge LP_{j-1}(s_{j-1})$ for $j > 1$. The predicate *link*(s, 1) is defined to be true for all s. We also use $s.x$ to refer to the value of the variable x in the state s.

The following lemma describes an assertion on the variable *curpred*.

LEMMA 3. *For all local states s:*

$$s.curpred = max \{j \mid link(s, j)\}$$

PROOF. We show that the above assertion is true for the initial state and is maintained by the program. Since *curpred* is initially 1, the assertion is trivially true for the initial state of any process. For the induction case, let $s \prec_{im} t$. We assume that the assertion holds for s, and show it to be true for t. we consider two cases:

Case 1: The event executed at s is not a receive

Since assertion holds for s, *link*(s, $s.curpred$) holds. Further, if $LP_{s.curpred}(s)$ is true then *link*(t, $s.curpred + 1$) holds as $s \prec_{im} t$. It is also easy to see that $s.curpred + 1$ is the maximum j such that *link*(t, j) holds. By incrementing $s.curpred$ the assertion is maintained for t.

Case 2: A message is received at s which was sent from the state u

This part of the proof follows from the observation that $link(s, j) \wedge s \to t$ implies that $link(t, j)$. If the assertion holds for s and u, then it is maintained by taking max of $s.curpred$ and $u.curpred$. □

THEOREM 4. *At the termination of the algorithm, there exists a process for which detectflag = true if and only if the* $\underline{A} : LP_1 \hookrightarrow LP_2 \hookrightarrow \cdots \hookrightarrow LP_m$ *is true.*

PROOF. We first show that if the strong linked predicate is true, then it is detected by the algorithm. Let the strong linked predicate be true. This means that at the termination there exists a state s such that *link*(s, $m + 1$) is true. From Lemma 3, $s.curpred = m + 1$. It follows that the process which has state s will have its detectflag set.

Conversely, assume that *s.detectflag* is true, i.e., $s.curpred = m + 1$. Again from Lemma 3, this means that *link*(s, $m + 1$) holds. From Theorem 2, this can happen only if $\underline{A} : LP_1 \hookrightarrow LP_2 \ldots \hookrightarrow LP_m$ is true. □

The above algorithm requires no extra messages but does require each message to contain the value of *curpred*. Hence, each message grows in size by O(log$m$) bits where m is the number of local predicates in the linked predicate.

The above algorithm can also be used to detect $\underline{A} : DP_1 \hookrightarrow DP_2 \hookrightarrow \cdots \hookrightarrow DP_m$, where each DP_i is a disjunction of local predicates. The only difference in detection of such a predicate from the strong linked predicate is that an index may occur in *pred_list* of more than one process.

Miller and Choi [20] have also proposed a similar algorithm for linked predicates. In their algorithm, a process p sends out a predicate marker along each channel directed away from p on detecting the local predicate. Thus, the algorithm assumes that underlying communication channels are FIFO. Note that this assumption is also exploited in stopping the program in a consistent state using an algorithm similar to that of Chandy and Lamport [5].

4 STRONG CONJUNCTIVE PREDICATES

Conjunctive predicates form the most interesting class of predicates in our logic. A strong conjunctive predicate is true if and only if the system will always reach a global state such that all of the given local predicates are true in that state. Formally, a strong conjunctive predicate is of the form: $\underline{A} : \Diamond (LP_1 \wedge ... \wedge LP_m)$, where LP_i for $1 \le i \le m$ are local predicates. Practically speaking, strong conjunctive predicates are most useful for good or desirable predicates (i.e., predicates which the programmer would like to be true at some point in the program). For example, in the case of a distributed two-phase commit protocol, if the master decides to commit a transaction, then it must be true that the program was in a global state where all the slaves were "ready" to commit. If the program is executed and commits, but a global state where all slave processes are "ready" does not occur, then the program has an error in it.

In this section, we present the conditions that are necessary and sufficient for a strong conjunctive predicate to hold. This is one of the main results of this paper. These conditions use the notion of *intervals*. An interval, I, is defined as a sequence of consecutive states of a trace having a beginning state (designated as $I.lo$) and an ending state (designated as $I.hi$). It is convenient to assume that $I.lo$ and $I.hi$ are distinct such that $I.lo \prec I.hi$. This is not a restriction. To model an interval with a single state it is sufficient to stutter that state once. A set of intervals, $I_1, ..., I_m$, each belonging to a different process trace is said to *overlap*, represented by, *overlap*(I_1, I_2, ... I_m), if and only if the following holds:

$$\forall i, j : i, j \in \{1 \ldots m\} : I_i.lo \to I_j.hi$$

Intuitively, the notion of overlapping intervals means that all the interval *los* are causally ordered before all the interval *his*.

We assume that $m \le n$ and $LP_1, \ldots, LP_m$ are local predicates in different processes, $P_1, \ldots, P_m$. (because $LP_1 \wedge LP_2$ is just another local predicate if LP_1 and LP_2 belong to the same process). We use $LP(I)$ to denote that the local predicate LP is true for the entire interval I.

The following lemma shows that existence of overlapping intervals is sufficient to ensure that all global sequences go through a global state in which $(LP_1 \wedge \ldots \wedge LP_m)$ is true. Intuitively, if two intervals I_1 and I_2 in processes P_1 and P_2 overlap, then it is not possible for P_1 to finish executing I_1 before P_2 enters the interval I_2 (and vice versa).

LEMMA 5. $\exists I_1, \ldots, I_m : LP_1(I_1) \wedge \ldots \wedge LP_m(I_m) \wedge overlap(I_1, \ldots, I_m) \Rightarrow \underline{A} : \Diamond(LP_1 \wedge \ldots \wedge LP_m)$.

PROOF. Using the definition for overlapping intervals we know that:

$$\forall i, j : i, j \in \{1 \ldots m\}: I_i.lo \rightarrow I_j.hi$$

This means that all *lo*s must appear before all *hi*s in any global sequence. Therefore, every possible global sequence has a state greater than or equal to all *lo*s and less than or equal to all *hi*s. In this state, the Boolean expression $LP_1 \wedge \ldots \wedge LP_m$ is true. Hence, the strong conjunctive predicate $\underline{A} : \Diamond\ (LP_1 \wedge \ldots \wedge LP_m)$ is true. □

We now show that these conditions are also necessary. Our obligation is to show that if these conditions are violated, then there exists a global sequence in which the strong conjunctive predicate is false. Our proof of the existence of such a global sequence is constructive. The global sequence we construct will have the property that it does not go through any global state in which all LP_i are true. We call such a global sequence *pure*. Formally,

DEFINITION 6. *A global sequence* $g = g_1, g_2, \ldots, g_m$ *is* pure *iff* $\forall k : \neg g_k \models LP_1 \wedge LP_2 \wedge \ldots \wedge LP_m$.

We will construct a pure global sequence by concatenating together multiple pure global subsequences. Let g be a global sequence of the run from a consistent global state x to a consistent global state y (i.e., x is the first global state in g and y is the last global state in g) and h be a global sequence from the global state y to a global state z. Then, it is easy to see that g concatenated with h is also a global sequence from x to z. In constructing a pure global sequence we use intermediate states which satisfy certain properties.

Let x be any global state. We denote by *first*(x) the m-tuple of intervals $(I_1(x), I_2(x), \ldots, I_m(x))$ where $I_k(x)$ is the first interval in $r[k]$ which ends after the state $x[k]$ in which LP_k is true. *first*(x) may not exist if for some process P_k, LP_k never becomes true after $x[k]$. A global state is called *consistent* if $\forall i, j : x[i] \,||\, x[j]$. We will use only consistent global states in our description. The intermediate (consistent) global states that we use to construct our pure global sequence satisfy an admissibility property.

DEFINITION 7. *x is an* admissible *intermediate global state if and only if*

1) *first*(x) *does not exist or*

2) $\exists k, l : I_k(x).lo \not\rightarrow I_l(x).hi \wedge \neg LP_k(x[k]) \wedge (\forall i$: $x[i] = \bot_i \vee (\exists j$: $x[i].prev \rightarrow x[j] \wedge x[j] = I.hi.next$ *for some interval* I))

The second disjunct says that there exists two intervals I_k and I_l such that they do not overlap $(I_k(x).lo \not\rightarrow I_l(x).hi)$ and LP_k is not true in the state $x[k]$. Further, x is *causally minimal* with respect to *I.hi.next* for some interval I.

Now, we are ready to show that:

LEMMA 8. $\neg\exists I_1, \ldots, I_m$: $LP_1(I_1) \wedge \ldots \wedge LP_m(I_m) \wedge overlap(I_1, \ldots, I_m) \Rightarrow \neg\underline{A} : \Diamond\ (LP_1 \wedge \ldots \wedge LP_m)$

PROOF. Let $\mathtt{I}(x)$ for any global state x be the set of all m-tuple of intervals in which LP_i is true for the ith trace (see Fig. 4) after the local state $x[i]$. We show that if none of these m-tuple of intervals satisfy overlapping condition, then there exists a global sequence in which the distributed program is never in any m-tuple in X.

Our aim is to construct a pure global sequence g from $start = (\bot_1, \bot_2, \ldots \bot_m)$ to $stop = (\top_1, \top_2, \ldots \top_m)$. Let x be any global state such that we have built a pure global sequence from *start* to x, and the remaining task is to build a pure global sequence from x to *stop*. Initially, we choose $x = start$. We will show a pure global sequence from x to y such that $|\mathtt{I}(y)| < |\mathtt{I}(x)|$. Thus, by continuing in this manner we will reach a global state z in which $|\mathtt{I}(z)| = 0$. From that point, all global sequences will be pure.

The *start* state is admissible, because by assumption either *first*(*start*) does not exist or there exist k, l such that $I_k(start).lo \not\rightarrow I_l(start).hi$. Moreover, LP_k is false in $\bot_k$ by (A2). Since $start[i] = \bot_i$, it is also causally minimal.

Now suppose that we are given an admissible global state x such that $|\mathtt{I}(x)| > 0$. For $1 \le j \le m$, let I_j be the first interval in $r[j]$ in which LP_j is true that ends after the state $x[j]$. As x is admissible and *first*(x) exists $(|\mathtt{I}(x)| > 0)$, there exist k, l such that $I_k(x).lo \not\rightarrow I_l(x).hi$ and $\neg\, LP_k(x[k])$. We define s to be the local state $I_l(x).hi.next$. The state s exists because of (A2) (it may be $\top_l$). We construct a global sequence from x to another admissible global state y, where y is defined as the minimum consistent global state after x such that $y[l] = s$. Such a global state exists because the set of all consistent cuts (ideals) is a lattice [18] and that ideals grow by adding one element at a time [8].

We first show that LP_k is never true between $x[k]$ and $y[k]$. It is sufficient to show that $y[k] \prec I_k(x).lo$. We know that $I_k(x).lo \not\rightarrow I_l(x).hi$. Applying (A1) twice, it follows that $I_k(x).lo.prev \not\rightarrow I_l(x).hi.next$. Thus, there exists a consistent global state z containing s such that $z[k] \le I_k(x).lo.prev$. As y is the minimum consistent global state with $y[l] = s$, we get that $y[k] \le I_k(x).lo.prev$. This implies that LP_k is never true between $x[k]$ and $y[k]$. Thus, all global sequences from x to y are pure. See Fig. 5.

We still need to show that y is admissible. If *first*(y) does not exist, we are done. Otherwise, we know that there exist k', l' such that $I_{k'}(y).lo \not\rightarrow I_{l'}(y).hi$ (see Fig. 5). If $\neg\, LP_{k'}(y[k'])$, then y is an admissible state and we are done. Otherwise, $y[k']$ is inside the interval $I_{k'}(y)$. We now do a case analysis. Since y is the causally minimal state with respect to s, it follows that $y[k'].prev \rightarrow s$ or $y[k'] = x[k']$.

Case 1: $y[k'].prev \rightarrow s$

From (A1) this is equivalent to $I_{k'}(y).lo \rightarrow I_l(x).hi$. We now show that y is admissible because $I_l(y).lo \not\rightarrow I_{l'}(y).hi \wedge \neg LP_l(y[l])$. The second conjunct is clearly true by the definition of y. We show the first conjunct. From $I_{k'}(y).lo \rightarrow I_l(x).hi$, and $I_l(x).hi < I_l(y).lo$ it follows that $I_{k'}(y).lo \rightarrow I_l(y).lo$. Therefore, $I_l(y).lo \rightarrow I_{l'}(y).hi$ is in contradiction with $I_{k'}(y).lo \not\rightarrow I_{l'}(y).hi$. Thus, the first unct $I_l(y).lo \not\rightarrow T_{l'}(y).hi$ also holds. Further, y it is easy to see that y is causally minimal.

Case 2: $y[k'] = x[k']$

Since x is admissible, there exists j such that $x[k'].prev \rightarrow x[j] \wedge x[j] = I.hi.next$. If $x[j] = y[j]$, then y is admissible because $I_j(y).lo \not\rightarrow I_{l'}(y).hi \wedge \neg LP_j(y[j])$. ($I_j(y).lo \rightarrow I_{l'}(y).hi$, $x[k'].prev \rightarrow x[j]$, $x[k'] = y[k']$, and $x[j] = y[j]$ contradict $I_{k'}(y).lo \not\rightarrow I_{l'}(y).hi$). If $x[j] \neq y[j]$, then $y[j].prev \rightarrow s$. From $y[k'] = x[k']$, $x[k'].prev \rightarrow x[j]$, $x[j] < y[j]$, $y[j].prev \rightarrow s$, we get $y[k'].prev \rightarrow s$. This is same as Case 1. □

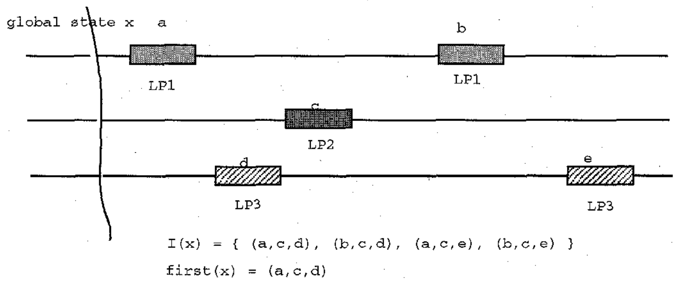


Fig. 4. I(*x*) and *first*(*x*).

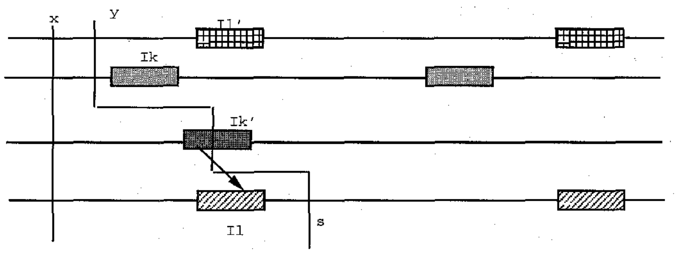


Fig. 5. Illustration of the proof of strong conjunctive predicates.

We see from the necessary and sufficient conditions for a strong conjunctive predicate to hold that the intervals delimited by *lo*s (local predicate transitioning from false to true) and *hi*s (transitions from true to false) must overlap.

At this point, we discuss the role of (A2). Consider a scenario in which two processes P_1 and P_2 are such that LP_1 and LP_2 are true throughout the execution of P_1 and P_2, respectively. If P_1 and P_2 never communicate with each other, then there does not exist overlapping intervals for LP_1 and LP_2. However, it may seem to the reader that for any global sequence there is a global state in which both LP_1 and LP_2 are true. The global sequence for which there does not exist any global state satisfying the strong conjunctive predicate is obtained by running one process to the completion before the other starts. Clearly, unless execution of both processes are synchronized in some manner, the above sequence is a proper global sequence. By (A2), LP_1 and LP_2 are false at the initial state (before the process has begun execution) and at final state (after the process has finished its execution).

4.1 Algorithms for Detecting a Strong Conjunctive Predicate

We now describe algorithms to check whether intervals in which local predicates hold overlap. These algorithms are executed by two kinds of processes: nonchecker processes and checker processes. They are based on a slight modification of timestamp vectors as proposed by Fidge [9] and Mattern [18]. Each process detects its local predicate and records the timestamp of the interval associated with the predicate. These intervals are sent to a checker process which uses them to decide if the strong conjunctive predicate became true.

Each nonchecker process (Fig. 6) keeps its own local *lcmvector* of timestamps. For process P_j, lcmvector[i] ($i \neq j$) is the message id of the last message from P_i (to anybody) which has a causal relationship to P_j. lcmvector[j] for process P_j is the next message id that P_j will use. Each time the local predicate of a process changes from false to true, the current value of lcmvector is remembered as an interval *lo*. At the next true-to-false transition (denoted by $\downarrow$ in the Fig. 6), the process sends the stored lcmvector (interval *lo*) and the current lcmvector (interval *hi*) to the checker process in a debug message. We next observe that a process is not required to send its interval every time the local predicate is detected. The interval need not be sent if there has been no message activity since the last time the interval was sent. This is because the lcmvector can change its value only when a message is sent or received. We now show that it is sufficient to send a lcmvector once after any message is received irrespective of the number of messages sent.

Let predicate *firstlmr(I)* be true iff the local predicate is true in I for the first time since the last message was received (or the beginning of the trace). We say $scp(I_1, I_2, ..., I_m)$ is true if $I_1, I_2, ..., I_m$ are the intervals in different processes making the strong conjunctive predicate true (as in Theorem 5).

THEOREM 9. $\exists I_1, ..., I_m : scp(I_1, I_2, ... I_m) \Rightarrow \exists J_1, ..., J_m, scp(J_1, J_2, ..., J_m) \wedge \forall k : 1 \leq k \leq m{:}firstlmr(J_k)$.

PROOF. By symmetry it is sufficient to prove the existence of J_1 such that $scp(J_1, I_2, ..., I_m) \wedge firstlmr(J_1)$. Let J_1 be the first interval in the trace of P_1 such that $LP(J_1)$ is true. Since $firstlmr(J_1)$ is true, our proof obligation is to show that $scp(J_1, I_2, ..., I_m)$. It is sufficient to show that $overlap(J_1, I_k)$ for $2 \leq k \leq m$. For any I_k, $I_1.lo \rightarrow I_k.hi$ and $J_1.lo \rightarrow I_1.lo$; therefore, $J_1.lo \rightarrow I_k.hi$. Also $I_k.lo \rightarrow I_1.hi$, because $overlap(I_k, I_1)$. Moreover, as there is no message received after $J_1.hi$ and before $I_1.hi$, the last causal message that made $I_k.lo \rightarrow I_1.hi$ true must have arrived before $J_1.hi$. Therefore, it is also true that $I_k.lo \rightarrow J_1.hi$.

Hence, we conclude that $overlap(J_1, I_k)$. □

```
Process P_id::
var
    lcmvector: array [1..n] of
      (0..MAXMID);
    init ∀i:i ≠ id: lcmvector[i] = 0,
      lcmvector[id] = 1;
        /* last causal msg rcvd from
          process 1 to n, respec. */
    Current_Interval: record lo, hi :
      (0..MAXMID);end;
    firstflag: Boolean init true;
    local_pred: Boolean_Expression;
      /*the local pred. to be tested by
        this process*/
□   For sending do
    send (prog, midgen, lcmvector, ...);
    lcmvector[id]++ ;
□   Upon receive (prog, mid,
    msg_lcmvector, ...) do
      ∀i: lcmvector[i] := max
        (lcmvector[i], msg_lcmvector[i]);
      firstflag := true;
□   Upon (local_pred ↑) ∧ firstflag do
      Current_Interval.lo := lcmvector;
□   Upon (local_pred ↓)∧ firstflag do
      Current_Interval.hi := lcmvector;
      send (dbg, Current_Interval) to
        CHECKERPROC;
      firstflag := false;
```

Fig. 6. Algorithm for strong conjunctive processes-checker process P_{id}.

The dominant space complexity of the above algorithm is due to the array "lcmvector" which is O(n). The main time complexity involves detecting the local predicates which is the same as for a sequential debugger. In the worst case, one debug message is generated for each program message received, so the worst case message complexity is O(m_r) where m_r is the number of program messages received.

We now give the algorithm for the checker process which detects the strong conjunctive predicate using the debug messages sent by other processes. The checker process has a separate queue for each process involved in the strong conjunctive predicate. Incoming debug messages from processes are enqueued in the appropriate queue. We ensure that the checker process gets its message from any process in a FIFO order. The required computation to check if the lcmvector u is less than the vector v in a different process is

$$(u[u.p] \leq v[u.p])$$

LEMMA 10. *Let I and J be intervals in processes P_i and P_j with vector pairs x and y, respectively. Then, $overlap(I, J)$ iff $(x.lo < y.hi) \wedge (y.lo < x.hi)$.*

PROOF. The proof follows from the fact that if s and t are states with time vectors u and v, then $s \rightarrow t$ iff $u < v$. See [18], [11]. □

Thus, the task of the checker process is reduced to checking ordering between lcmvectors to determine if the intervals overlap. Because of the above Lemma, we use terms intervals and vector-pairs interchangeably. The following Lemma shows how the checker process can avoid checking all possible combinations of intervals.

LEMMA 11. *Let x and y be two vector pairs at the head of their respective queues. If they do not overlap, then at least one of them can be eliminated from further consideration in checking to see if the strong conjunctive predicate is satisfied.*

PROOF. In order for the strong conjunctive predicate to be true, there must exist a set of intervals, one from each queue, such that each overlaps with all the others in the set. Let two intervals x and y be at the head of their queues such that they do not overlap. This means that either $x.lo \not< y.hi$ or $y.lo \not< x.hi$. Assume the former without any loss of generality. We show that y can be eliminated in this case. If not, let x' be another interval in the queue of x which overlaps with y. This implies that $x'.lo \rightarrow y.hi$. Since $x.lo \rightarrow x'.lo$, we conclude that $x.lo \rightarrow y.hi$, a contradiction. □

The checker process receives debug messages containing timestamp pairs from the other processes and executes the algorithm in Fig. 7. Each element of the queue is an interval, and the comparisons are done between *his* and *los* of these intervals. The checker process reduces the number of comparisons by deleting any vector-pair at the head of any queue whose *hi* lcmvector is not greater than *lo* lcmvector of vector-pairs of head of all other queues. The checker process has detected the strong conjunctive predicate to be true if it finds a set of intervals at the head of queues such that they are pairwise overlapping.

```
var
    q_1 ... q_m: queue of record lo, hi:
      timevector;end;
    changed, newchanged: set of
      {1, 2, ..., m}
□   Upon recv(elem) from P_k do
      insert(q_k, elem);
      if (head(q_k) = elem) then begin
        changed := {k};
        while (changed ≠ φ) begin
          newchanged := {};
          for i in changed, and j in
            [1, 2, ..., m] do begin
              if head(q_j).lo ≮ head(q_i).hi
                then newchanged:=newchanged
                ∪ {i};
              if head(q_i).lo ≮ head(q_j).hi
                then newchanged:=newchanged
                ∪ {j};
          end; /* for */
          changed := newchanged;
          for i in changed do
            deletehead(q_i);
        end; /* while */
        if ∀i: ¬ empty(q_i) then
          found:=true;
      end; /* if */
```

Fig. 7. Algorithm for strong conjunctive checker process.

This algorithm requires at most $O(m^2p)$ comparisons where m is the number of queues each of length at most p.

5 Decentralization of the Algorithm

We now show techniques for decentralizing the above algorithm. If a set of intervals S is such that all pairs of intervals overlap, then the following holds:

$$\forall x, y \in S : x.lo < y.hi \qquad (P1)$$

We denote this by predicate *overlap*(S). Our aim is to show that the above condition can be checked in a decentralized manner. For this, we need the concept of greatest lower bound of a set of intervals. Let X be set of all intervals, where each interval x is defined as a pair of vectors $x.lo$ and $x.hi$ such that $x.lo \le x.hi$. We now define an order $\sqsubseteq$ between elements in this set as follows:

$$x \sqsubseteq y \equiv (x.lo \ge y.lo) \wedge (x.hi \le y.hi)$$

It can be easily checked that $(\mathrm{X}, \sqsubseteq)$ is a partial order. In this partial order, $x \sqcap y = (max(x.lo, y.lo), min(x.hi, y.hi))$. Then,

$$overlap(x, y) \Rightarrow (x \sqcap y) \in \mathrm{X}$$

Further, if $x_1, x_2, \ldots, x_m$ are such that $\forall i, j$: $overlap(x_i, x_j)$, then

$$\sqcap_i x_i \in \mathrm{X}$$

The following theorem shows that the process of finding *overlap*(X) can be decomposed into smaller sets.

Theorem 12. Let X, Y, and Z be sets of intervals, such that $X = Y \cup Z$. Then, $overlap(X)$ iff $overlap(Y) \wedge overlap(Z) \wedge overlap(\sqcap\{x \mid x \in Y\}, \sqcap\{x \mid x \in Z\})$.

Proof. ($\Rightarrow$) $overlap(Y)$ and $overlap(Z)$ are clearly true because $Y, Z \subseteq X$. We need to show that

$$overlap(\sqcap\{x \mid x \in Y\}, \sqcap\{x \mid x \in Z\})$$

Let $y^* = \sqcap\{x \mid x \in Y\}$, and $z^* = \sqcap\{x \mid x \in Z\}$. Since $overlap(Y)$ and $overlap(Z)$, y^* and z^* belong to X. To prove $overlap(y^*, z^*)$, we need to show that $(y^*.lo < z^*.hi) \wedge (z^*.lo < y^*.hi)$. We show just the first conjunct.

From $overlap(X)$, we get that

$$\forall y, z \in X : y.lo < z.hi.$$

In particular,

$$\forall y \in Y, z \in Z: y.lo < z.hi.$$

Then, by definition of y^* and z^*, we conclude that

$$y^*.lo < z^*.hi$$

($\Leftarrow$) We show that ($P1$) holds for X, i.e.,

$$\forall y, z \in X : y.lo < z.hi$$

If both y and z belong either to Y and Z, then the above is true from $overlap(Y)$ and $overlap(Z)$. Let us assume without loss of generality that $y \in Y$ and $z \in Z$. We need to show that $y.lo < z.hi$. This is true because $y.lo \le y^*.lo < z^*.hi \le z.hi$. The first and the last inequality follow from the definition of y^* and z^*; the middle inequality follows from $overlap(y^*, z^*)$. □

Using the above theorem and the notions of a hierarchy, the algorithm for checking the strong conjunctive predicate can be decentralized as follows. We may divide the set of processes into two groups. The group checker process checks for the strong conjunctive predicate within its group. On finding one, it sends the greatest lower bound of all intervals to a higher process in the hierarchy. This process checks the last conjunct of the above theorem. Clearly, the above argument can be generalized to a hierarchy of any depth.

6 Applications

The main application of our results are in debugging and testing of distributed programs. We have incorporated our algorithms in a distributed debugger [15]. The on-line debugger is able to detect global states or sequences of global states in a distributed computation. The architecture of this distributed debugger is shown in Fig. 8. With each application process, we attach two processes: a *gdb* process and a *monitor* process. *gdb* is a sequential debugger that we use for detecting local predicates. *monitor* processes are responsible for attaching vector time information with all messages. They also report to the centralized *coordinator* process whenever an interval is detected. Monitor processes also detect strong linked predicates using the algorithm outlined earlier. There is one *coordinator* process in the system. It receives all the information from monitor processes and checks for strong and weak conjunctive predicates. The coordinator also provides a single user-interface to the programmer. Our distributed debugger runs on a cluster of Sun workstations running SunOS.

We have also used our algorithms to implement a trace analyzer for distributed programs [6]. Our analyzer monitors a distributed program and gathers enough information to form a distributed run. The user can then ask whether any global predicate became true.

7 Conclusions

We have discussed detection of global predicates in a distributed program. Earlier algorithms for detection of global predicates proposed by Chandy and Lamport work only for stable predicates. Our algorithms detect even unstable predicates with reasonable time, space and message complexity.

In this paper, we have emphasized conjunctive predicates and not disjunctive predicates. The reason is that disjunctive predicates are quite simple to detect. Disjunctive predicates are of the form $\underline{A} : LP_1 \vee LP_2 \vee \ldots \vee LP_m$, or of the form $\underline{E} : LP_1 \vee LP_2 \vee \ldots \vee LP_m$. It turns out that for the simple case considered here, both expressions are equivalent. To detect a disjunctive predicate $\underline{A} : LP_1 \vee LP_2 \vee \ldots \vee LP_m$, it is sufficient for process P_i to monitor LP_i. If any of the process finds its local predicate true, the disjunctive predicate is true.

We have also not discussed predicates of the form $\underline{A} : \Box bool$. These predicates are duals of $\underline{E} : \Diamond bool$ which have been discussed in [12].

Algorithms given in this paper detect predicates of the form $\underline{A} : \Diamond bool$, where *bool* is a conjunction of local predicates. It would be of great interest if these algorithms can be generalized to detect predicates when *bool* is any Boolean

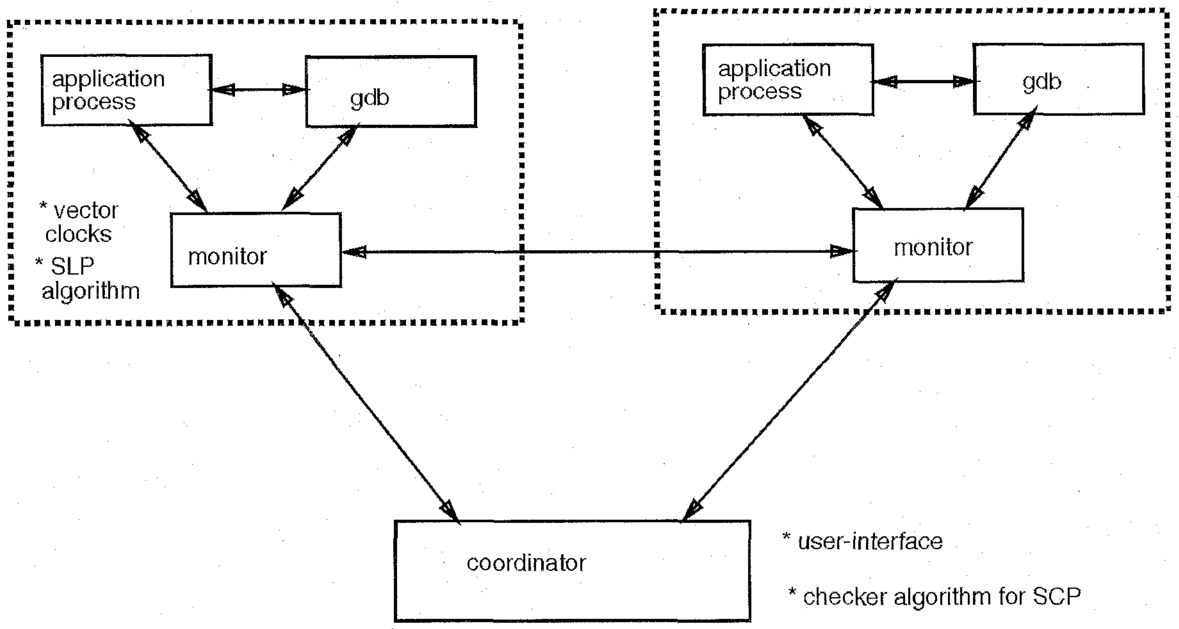


Fig. 8. Architecture of our distributed debugger.

expression of local predicates.

Acknowledgments

We would like to thank Bryan Chin, Mohamed Gouda, Greg Hoagland, Jay Misra, Michel Raynal, Scott Stoller, and Alex Tomlinson for their comments on earlier versions of this work. We would also like to thank Bryan Chin for implementing off-line versions of our algorithms and Greg Hoagland for incorporating our algorithms in a distributed debugger. We also thank the anonymous referees for their meticulous review of an earlier version of the paper.

This work was supported in part by the U.S. National Science Foundation Grant CCR 9110605, a TRW faculty assistantship award, a General Motors fellowship, and an IBM grant.

References

[1] O. Babaoglu and K. Marzullo, "Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms," *Distributed Systems*, second edition, S. Mullender, ed., ACM Press, Frontier Series, 1993.

[2] P. Bates, "Distributed Debugging Tools for Heterogeneous Distributed Systems," *Proc. Eighth Int'l Conf. Distributed Computing Systems*, pp. 308–315, 1988.

[3] K.P. Birman and T.A. Joseph, "Reliable Communication in the Presence of Failures," *ACM Trans. Computer Systems*, vol. 5, no. 1, pp. 47–76, 1987.

[4] L. Bouge, "Repeated Snapshots in Distributed Systems with Synchronous Communication and Their Implementation in CSP," *Theoretical Computer Science*, vol. 49, pp. 145-169, 1987.

[5] K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans. Computer Systems*, pp. 63–75, Feb. 1985.

[6] B. Chin, "An Offline Debugger for Distributed Programs," MS thesis, Electrical and Computer Eng. Dept., Univ. of Texas at Austin, Dec. 1991.

[7] R. Cooper and K. Marzullo, "Consistent Detection of Global Predicates," *Proc. ACM Workshop Parallel and Distributed Debugging*, pp. 163–173, Santa Cruz, Calif., May 20–21, 1991.

[8] C. Diehl, C. Jard, and J. Rampon, "Reachability Analysis on Distributed Executions," *Proc. TAPSOFT '93: Theory and Practice of Software Development*, pp. 629–643, Orsay, France, Apr. 13–17, 1993.

[9] C. Fidge, "Partial Orders for Parallel Debugging," *Proc. ACM Workshop Parallel and Distributed Debugging*, pp. 130–140, Madison, Wis., May 1988.

[10] V.K. Garg and B. Waldecker, "Detection of Unstable Predicate in Distributed Programs," *Proc. 12th Conf. Foundations Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science 652, pp. 253-264, Springer-Verlag, Dec. 1992.

[11] V.K. Garg and A.I. Tomlinson, "Using Induction to Prove Properties of Distributed Programs," *Proc. Symp. Parallel and Distributed Processing*, pp. 478–485, Dallas, Dec. 1993.

[12] 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, Mar. 1994.

[13] V.K. Garg and B. Waldecker, "Detection of Strong Unstable Predicates in Distributed Programs," technical report, Electrical and Computer Eng. Dept., Univ. of Texas at Austin, 1994.

[14] D. Haban and W. Weigel, "Global events and global breakpoints in distributed systems," *Proc. 21st Int'l Conf. System Sciences*, vol. 2, pp. 166–175, Jan. 1988.

[15] G. Hoagland, "A Debugger for Distributed Programs," MS thesis, Electrical and Computer Eng. Dept., Univ. of Texas at Austin, Aug. 1992.

[16] M. Hurfin, N. Plouzeau, and M. Raynal, "Detecting Atomic Sequences of Predicates in Distributed Computations," *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, San Diego, Calif., pp. 32–42, May 1993.

[17] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM*, pp. 558–565, July 1978.

[18] 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, Elsevier Science Publishers B.V., 1989.

[19] C.E. McDowell and D.P. Helmbold, "Debugging Concurrent Programs," *ACM Computing Surveys*, vol. 21, no. 4, pp. 593–622, Dec. 1989.

[20] B.P. Miller and J. Choi, "Breakpoints and Halting in Distributed Programs," *Proc. Eighth Int'l Conf. Distributed Computing Systems*, 1988.

[21] R. Schwartz and F. Mattern, "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail," SFB124-15/92, Dept. of Computer Science, Univ. of Kaiserslautern, Germany, Dec. 1992.

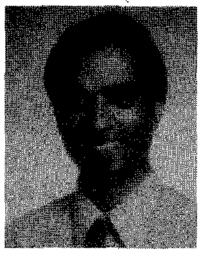
[22] M. Spezialetti and P. Kearns, "Efficient Distributed Snapshots," *Proc. Sixth Int'l Conf. Distributed Computing Systems*, pp. 382–388, 1986.

[23] M. Spezialetti and P. Kearns, "Simultaneous Regions: A Framework for Consistent Monitoring of Distributed Systems," *Proc. Ninth Int'l Conf. Distributed Computing Systems*, pp. 61–68, 1988.

[24] A.I. Tomlinson and V.K.Garg, "Detecting Relational Global Predicates in Distributed Systems," *Proc. Third ACM/ONR Workshop Parallel and Distributed Debugging*, San Diego, Calif., pp. 21–31, May 1993.

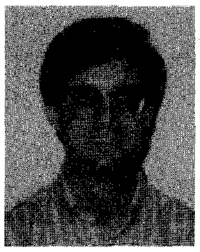
[25] S. Venkatesan and B. Dathan, "Testing and Debugging Distributed Programs Using Global Predicates," *Proc. 30th Ann. Allerton Conf. Comm., Control, and Computing*, pp. 137–146, Allerton, Ill., Oct. 1992.

[26] B. Waldecker and V.K. Garg, "Unstable Predicate Detection in Distributed Programs," *Proc. Second ACM/ONR Workshop Parallel and Distributed Debugging*, extended abstract, Santa Cruz, Calif., 1991.



Vijay K. Garg (SM'84-M'89-SM'96) received his Bachelor of Technology degree in computer engineering from the Indian Institute of Technology, Kanpur, in 1984; and his MS and PhD degrees in electrical engineering and computer science from the University of California, Berkeley, in 1985 and 1988, respectively. He is now an associate professor in the Department of Electrical and Computer Engineering at the University of Texas, Austin, and director of the Parallel and Distributed Systems Laboratory (PDSLAB).

Dr. Garg's research interests are in the areas of distributed systems and discrete event systems. He is the author of the book *Principles of Distributed Systems* and a coauthor of the book *Modeling and Control of Logical Discrete Event Systems* (Kluwer Academic Publishers). He has served as a program committee member of the IEEE International Conference on Distributed Computing Systems and as an organizer of the minisymposium on Discrete Event Systems at the SIAM Conference on Control and Applications. Prof. Garg received the 1992 TRW faculty assistantship award and holds the General Motor Centennial Fellowship in Electrical Engineering. He is a senior member of the IEEE.



Brian Waldecker received the BSEE and BA degrees in computer science degrees from Rice University in Houston, Texas, in 1986, and the MS and PhD in electrical and computer engineering from the University of Texas at Austin, in 1988 and 1991, respectively. He is currently with IBM, Austin. His interests include distributed computing systems and distributed program behavior. He is a member of the IEEE Computer Society, the ACM, and the Society of Petroleum Engineers.