Detection of Weak Unstable Predicates in Distributed Programs

Vijay K. Garg, Brian Waldecker

Abstract— **This paper discusses 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. A predicate is stable if it does not turn false once it becomes true. Our algorithms detect even unstable predicates without excessive overhead. In the past, such predicates have been regarded as too difficult to detect. The predicates are specified using a logic described formally in this paper. We discuss detection of weak conjunctive predicates which are formed by conjunction of predicates local to processes in the system. Our detection methods will detect if such a predicate is true for any interleaving of events in the system, whether the predicate is stable or not. Also, any predicate which can be reduced to a set of weak conjunctive predicates is detectable. This class of predicates captures many global predicates that are of interest to a programmer. The message complexity of our algorithm is bounded by the number of messages used by the program. The main application of our results are in debugging and testing of distributed programs. Our algorithms have been incorporated in a distributed debugger which runs on a network of Sun Workstations under SunOS.**

Keywords— **Unstable Predicates, Predicate Detection, Distributed Algorithms, Distributed Debugging**

I. Introduction

A distributed program is one that runs on multiple processors connected by a communication network. The state of such a program is distributed across the network and no process has access to the global state at any instant. Detection of a global predicate, i.e. a condition that depends on the state of multiple processes, is a fundamental problem in distributed computing. This problem arises in many contexts such as designing, testing and debugging of distributed programs.

A global predicate may be either stable or unstable. A stable predicate is one which never turns false once it becomes true. Some examples of stable predicates are deadlock and termination. Once a system has terminated it will stay terminated. An unstable predicate is one without such a property. Its value may alternate between true and false. Chandy and Lamport [3] have given an elegant algorithm to detect stable predicates. Their algorithm is based on taking a consistent global snapshot of the system and checking if the snapshot satisfies the global predicate. If the snapshot satisfies the stable predicate, then it can be inferred that the stable predicate is true at the end of the snapshot algorithm. Similarly, if the predicate is false for the snapshot, then it was also false at the beginning of the snapshot algorithm. By taking such snapshots periodically a stable property can be detected. Bouge [2] , and Spezialetti and Kearns [22] have extended this method for repeated snapshots. This approach does not work for unstable predicate which may be true only between two snapshots and not at the time when the snapshot is taken. An entirely different approach is required for such predicates.

In this paper, we present an approach which detects a large class of unstable predicates. We begin by defining a logic that is used for specification of global predicates. Formulas in this logic are interpreted over a *single* run of a distributed program. A run of a distributed program generates a partial order of events, and there are many total orders consistent with this partial order. We call a formula strong if it is true for all total orders, and weak if there exists a total order for which it is true. We consider a special class of predicates defined in this logic in which a global state formula is either a disjunction, or a conjunction of local predicates. Since disjunctive predicates can simply be detected by incorporating a local predicate detection mechanism at each process, we focus on conjunctive predicates. In this paper, we describe algorithms for detection of weak types of these predicates. Detection of strong predicates is discussed in [10] .

Many of our detection algorithms use timestamp vectors as proposed by Fidge [6] and Mattern [17]. Each process detects its local predicate and records the timestamp associated with the event. These timestamps are sent to a checker process which uses these timestamps to decide if the global predicate became true. We show that our method uses the optimal number of comparisons by providing an adversary argument. We also show that the checking process can be decentralized, making our algorithms useful even for large networks.

The algorithms presented in this paper have many applications. In debugging a distributed program, a programmer may specify a breakpoint on a condition using our logic and then detect if the condition became true. Our algorithms can also be used for testing distributed programs. Any condition that must be true in a valid run of a distributed program may be specified and then its occurrence can be verified. An important property of our algorithms is that they detect even those errors which may not manifest themselves in a particular execution, but may do so with different processing speeds. As an example, consider

This work was supported in part by the NSF Grant CCR 9110605, the Navy Grant N00039-91-C-0082, a TRW faculty assistantship award, and IBM Agreement 153.

V.K. Garg is with the Electrical and Computer Engineering Dept, University of Texas at Austin, Austin, TX 78712-1084. vijay@pine.ece.utexas.edu B. Waldecker is with Austin System Center of Schlumberger Well Services, Austin, TX 78720.

a distributed mutual exclusion algorithm. In some run, it may be possible that two processes do not access critical region even if they both had permission to enter the critical region. Our algorithms will detect such a scenario under certain conditions described in the paper.

Cooper and Marzullo [5], and Haban and Weigel [11] also describe predicate detection, but they deal with general predicates. Detection of such predicates is intractable since it involves a combinatorial explosion of the state space. For example, the algorithm proposed by Cooper and Marzullo [5] 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. Miller and Choi [19] discuss mainly linked predicates. They do not discuss detection of conjunctive predicates (in our sense) which are most useful in distributed programs. Moreover, they do not make distinction between *program* messages and messages used by the detection algorithm. As a result, the linked predicate detected by Miller and Choi's algorithm may be true when the debugger is present but may become false when it is removed. Our algorithms avoid this problem. Hurfin, Plouzeau and Raynal [12] also discuss methods for detecting atomic sequences of predicates in distributed computations. Spezialetti and Kearns [23] discuss methods for recognizing event occurrences without taking snapshots. However, their approach is suitable only for monotonic events which are similar to stable properties. An overview of these and some other approaches can be found in [20].

This paper is organized as follows: Section II presents our logic for describing unstable predicates in a distributed program. It describes the notion of a distributed run, a global sequence and the logic for specification of global predicates. Section III discusses a necessary and sufficient condition for detection of weak conjunctive predicates. It also shows that detection of weak conjunctive predicates is sufficient to detect any global predicate on a finite state program, or any global predicate that can be written as a boolean expression of local conditions. Section IV presents an algorithm for detection of a weak conjunctive predicate. Section V describes a technique to decentralize our algorithm. Section VI gives some details of an implementation of our algorithms in a distributed debugger. Finally, section VII gives conclusions of this paper.

II. Our Model

A. Distributed Run

We assume a loosely-coupled message-passing system without any shared memory or a global clock. A distributed program consists of a set of n processes denoted by $\{P_1, P_2, ..., P_n\}$ communicating solely via asynchronous messages. In this paper, we will be concerned with a single run r of a distributed program. Each process P_i in that run generates a single execution trace $r[i]$ which is a finite sequence of *states* and *actions* which alternate beginning with an initial state. The state of a process is defined by the value of all its variables including its program counter. For example, the process P_i generates the trace $s_{i,0}a_{i,0}s_{i,1}a_{i,1}\ldots a_{i,l-1}s_{i,l}$, where s_i's are the local states and a_i's are the local actions in the process P_i. There are three kinds of actions - internal, send and receive. A send action denoted by $send(<i,j,\phi>)$ means the sending of a message ϕ from the process P_i to the process P_j. A receive action denoted by $receive(<i,j,\phi>)$ means the receiving of a message ϕ from the process P_i by the process P_j. We assume in this paper that no messages are lost, altered or spuriously introduced. We do not make any assumptions about FIFO nature of the channels. A *run* r is a vector of traces with $r[i]$ as the trace of the process P_i. From the reliability of messages we obtain

$$receive(<i,j,\phi>) \in r[j] \Leftrightarrow send(<i,j,\phi>) \in r[i]$$

We also define a happened-before relation (denoted by $\rightarrow$) between states similar to that of Lamport's happened-before relation between events .

Definition 1 *The state s in the trace $r[i]$ happened-before ($\rightarrow$) the state t in the trace $r[j]$ if and only if one of the following holds:*
1. $i = j$ and s occurs before t in $r[i]$.
2. The action following s is the send of a message and the action before t is the reception of that message.
3. There exists a state u in one of the traces such that $s \rightarrow u$ and $u \rightarrow t$.

The relation $\rightarrow$ is a partial order on the states of the processes in the system. As a result of rules 2 and 3 in the above definition, we say that there is a *message path* from state s to state t if $s \rightarrow t$ and they are in different processes. A run can be visualized as a valid error-free process time diagram [16].

Example 2 Consider the following distributed program:

Process P_1;
var x:integer initially 7;
begin
l_0: send(x) to P_2;
l_1: x := x-1;
l_2: send(x) to P_2;
l_3:
end;
	Process P_2;
	var y,z:integer initially (0,0);
	begin
	m_0: receive(y) from P_1;
	m_1: receive(z) from P_1;
	m_2:
	
	end;

Labels $l_0, ..., l_3$ and $m_0, ..., m_2$ denote possible values of program counters. A distributed run r is given by:
$r[1] = ((l_0,7), send(1,2,7), (l_1,7), internal,$
$(l_2,6), send(1,2,6), (l_3,6))$
$r[2] = ((m_0,0,0), receive(1,2,7), (m_1,7,0),$
$receive(1,2,6), (m_2,7,6))$
Another run r' can be constructed when two messages sent by the process P_1 are received in the reverse order.
$r'[1] = ((l_0,7), send(1,2,7), (l_1,7), internal,$
$(l_2,6), send(1,2,6), (l_3,6))$
$r'[2] = ((m_0,0,0), receive(1,2,6), (m_1,6,0),$
$receive(1,2,7), (m_2,6,7))$

B. Global Sequence

A run defines a partial order ($\rightarrow$) on the set of actions and states. For simplicity, we ignore actions from a run and focus just on states in traces. Thus, $r[i]$ denotes the sequence of states of P_i. In general, there are many total orders that are consistent with (or linearizations of) this partial order. A *global sequence* corresponds to a view of the run which could be obtained given the existence of a global clock. Thus, a global sequence is a sequence of global states where a global state is a vector of local states. This definition of a global state is different from that of Chandy and Lamport which includes states of the channels. In our model, a channel is just a 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. We denote the set of global sequences consistent with a run r as $linear(r)$. A *global sequence* g is a finite sequence of global states denoted as $g = g_0 g_1 \ldots g_l$, where g_k is a global state for $0 \leq k \leq l$. Its suffix starting with g_k (i.e., $g_k.g_{k+1} \ldots g_l$) is denoted by g^k. Clearly, 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. The stutter arises because we have purposely avoided any reference to physical time. Let $s||t$ mean that $s \not\rightarrow t \wedge t \not\rightarrow s$. Then, a global sequence of a run is defined as:

Definition 3 *g is a global sequence of a run r (denoted by $g \in linear(r)$) 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): $\forall k : g_k[i] || g_k[j]$ where $g_k[i]$ is the state of P_i in the global state g_k

Example 4 *Some global sequences consistent with the run r in Example 2 are given below:*
$g = [(l_0, 7, m_0, 0, 0), (l_1, 7, m_0, 0, 0), (l_2, 6, m_1, 7, 0),$
$(l_3, 6, m_1, 7, 0), (l_3, 6, m_2, 7, 6)]$
$h = [(l_0, 7, m_0, 0, 0), (l_1, 7, m_0, 0, 0), (l_2, 6, m_0, 0, 0),$
$(l_3, 6, m_1, 7, 0), (l_3, 6, m_2, 7, 6)]$

Our model of a distributed run and global sequences does not assume that the system computation can always be specified as some interleaving of local actions. The next global state of a global sequence may result from multiple independent local actions.

C. Logic Operators

There are three syntactic categories in our logic - *bool*, *lin* and *form*. The syntax of our logic is as follows:

form	::=	$\underline{\text{A}}$: lin | $\underline{\text{E}}$: lin
lin	::=	$\Box$ lin | $\Diamond$ lin | lin $\hookrightarrow$ lin | lin $\wedge$ lin | lin $\vee$ lin | $\neg$lin | bool
bool	::=	a predicate over a global system 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 $(x = 7, y = 0)$, then the *bool* $(x \geq y)$ is true. Here x and y could be variables in different processes. A *lin* is a temporal formula defined over a global sequence. $\Diamond$ *lin* means that there exists a suffix of the global sequence such that *lin* is true for the suffix [21]. We also use $\Box$ as the dual of $\Diamond$. We have also introduced a binary operator ($\hookrightarrow$) to capture sequencing directly. $p \hookrightarrow q$ means that there exists suffixes g^i and g^j of the global sequence such that p is true of the suffix g^i, q is true of the suffix g^j, and $i < j$. A *form* is defined over a set of global sequences and it is simply a *lin* qualified with the universal ($\underline{\text{A}}$:) or the existential ($\underline{\text{E}}$:) quantifier. Thus, the semantics of our logic is as follows:

$g \models bool$	**iff**	$g_0 \models bool$
$g \models \neg lin$	**iff**	$\neg(g \models lin)$
$g \models lin_1 \wedge lin_2$	**iff**	$g \models lin_1 \ \wedge g \models lin_2$
$g \models lin_1 \vee lin_2$	**iff**	$g \models lin_1 \ \vee g \models lin_2$
$g \models \Box lin$	**iff**	$\forall i : g^i \models lin$
$g \models \Diamond lin$	**iff**	$\exists i : g^i \models lin$
$g \models lin_1 \hookrightarrow lin_2$	**iff**	$\exists i, j : (i < j) \wedge g^i \models lin_1 \wedge g^j \models lin_2$
$r \models \underline{\text{A}}{:}lin$	**iff**	$\forall g : g \in linear(r) : g \models lin$
$r \models \underline{\text{E}}{:}lin$	**iff**	$\exists g : g \in linear(r) : g \models lin$

$\underline{\text{A}}$, and $\underline{\text{E}}$ quantify over the set of global sequences that a distributed run may exhibit given the trace for each process. $\underline{\text{A}}$:p means that the predicate p holds for all global sequences and $\underline{\text{E}}$:p means that the predicate p holds for some global sequence. We call formulas starting with $\underline{\text{A}}$: as *strong* formulas and formulas starting with $\underline{\text{E}}$: as *weak* formulas. The intuition behind the term *strong* is that a strong formula is true no matter how fast or slow the individual processes in the system execute. That is, it holds for all execution speeds which generate the same trace for an individual process. A *weak* formula is true if and only if there exists 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.

The difficulty of checking truthness of a global predicate arises from two sources. First, if there are n processes in the system, the total number of global sequences (in which a global state is not repeated) is exponential in n and the size of the traces. Secondly, the global state is distributed across the network during an actual run. Thus, detection of any general predicate in the above logic is not feasible in a distributed program. To avoid the problem of combinatorial explosion, we focus on detection of predicates belonging to a class that we believe captures a large subset of predicates interesting to a programmer. We use the word *local* to refer to a predicate or condition that involves the state of a single process in the system. Such a condition can be easily checked by the process itself. We detect predicates that are boolean expressions of local predicates. Following are examples of the formulas detectable by our algorithms:

1. Suppose we are developing a mutual exclusion algorithm. Let CS_i represent the local predicate that the process P_i is in critical section. Then, the following formula detects any possibility of violation of mutual exclusion for

a particular run:

$$\underline{E} : \Diamond(CS_1 \wedge CS_2)$$

2. In the example 4, we can check if

$$\underline{E} : \Diamond(x = 6) \wedge (P_2\ at\ m_0)$$

Note that

$$\Diamond(x = 6) \wedge (P_2\ at\ m_0)$$

is not true for the global sequence g, but it is true for the global sequence h. Our algorithm will detect the above predicate to be true for the run r even though the global sequence executed may be g.

3. Assume that in a database application, serializability is enforced using a two phase locking scheme [15]. Further assume that there are two types of locks: *read* and *write*. Then, the following formula may be useful to identify an error in implementation:

$$\underline{E} : \Diamond(P_1\ has\ read\ lock) \wedge (P_2\ has\ write\ lock)$$

III. Weak Conjunctive Predicates

A weak conjunctive predicate (WCP) is true for a given run if and only if there exists a global sequence consistent with that run in which all conjuncts are true in some global state. Practically speaking, this type of predicate is most useful for bad or undesirable predicates (i.e. predicates that should never become true). In such cases, the programmer would like to know whenever it is *possible* that the bad predicate may become true. As an example, consider the classical mutual exclusion situation. We may use a WCP to check if the correctness criterion of never having two or more processes in their critical sections at the same time is met. We would want to detect the predicate "process x is in its critical section *and* process y is in its critical section". It is important to observe that our algorithms will report the possibility of mutual exclusion violation even if it was not violated in the execution that happened. The detection will occur if and only if there exists a consistent cut in which all local predicates are true. Thus, our techniques detect errors that may be hidden in some run due to race conditions.

A. Importance of Weak Conjunctive Predicates

Conjunctive predicates form the most interesting class of predicates because their detection is sufficient for detection of any global predicate which can be written as a boolean expression of local predicates. This observation is shown below:

Lemma 5 *Let p be any predicate constructed from local predicates using boolean connectives. Then, $\underline{E} : \Diamond p$ can be detected using an algorithm that can detect $\underline{E} : \Diamond q$ where q is a pure conjunction of local predicates.*

Proof: We first write p in its disjunctive normal form. Thus, $\underline{E}$: $\Diamond$ p = $\underline{E}$: $\Diamond$ $(m_1 \vee \ldots \vee m_l)$ where each m_i is a pure conjunction of local predicates. Next, we observe that

$\quad \underline{E}$: $\Diamond$ $(m_1 \vee \ldots \vee m_l)$
= { semantics of $\underline{E}$ and $\Diamond$ }
$\quad \exists g : \exists i : g^i \models (m_1 \vee m_2 \ldots \vee m_l)$
= { semantics of $\vee$}
$\quad \exists g : \exists i : (g_i \models m_1 \vee g_i \models m_2 \vee \ldots g_i \models m_l)$
= { distribute $\exists$ over $\vee$ twice}
$\quad \exists g : \exists i : g_i \models m_1\ \vee\ \ldots\ \vee\ \exists g : \exists i : g_i \models m_l$
= { semantics of $\underline{E}$ and $\Diamond$ }
$\quad \underline{E}$: $\Diamond m_1\ \vee\ \underline{E}$:$\Diamond m_2 \vee\ \ldots\ \vee\ \underline{E}\Diamond m_l$

Thus, the problem of detecting $\underline{E} : \Diamond p$ is reduced to solving l problems of detecting $\underline{E} : \Diamond q$ where q is a pure conjunction of local predicates.

■

Our approach is most useful when the global predicate can be written as a boolean expression of local predicates. As an example, consider a distributed program in which x, y and z are in three different processes. Then,
$\underline{E}$:$\Diamond even(x) \wedge ((y < 0) \vee (z > 6))$
can be rewritten as
$\underline{E}$:$\Diamond(even(x) \wedge (y < 0)) \vee$ $\underline{E}$:$\Diamond(even(x) \wedge (z > 6))$
where each part is a weak conjunctive predicate.

We note that even if the global predicate is not a boolean expression of local predicates, but it is satisfied by only a finite number of possible global states, then it can again be rewritten as a disjunction of weak conjunctive predicates. For example, consider the predicate $\underline{E} : \Diamond(x = y)$, where x and y are in different processes. $(x = y)$ is not a *local* predicate as it depends on both processes. However, if we know that x and y can only take values $\{0, 1\}$, then the above expression can be rewritten as
$\underline{E} : \Diamond((x = 0) \wedge (y = 0)) \vee ((x = 1) \wedge (y = 1))$.
This is equivalent to
$(\underline{E} : \Diamond(x = 0) \wedge (y = 0)) \vee (\underline{E} : \Diamond(x = 1) \wedge (y = 1))$.
Each of the disjunct in this expression is a weak conjunctive predicate.

We observe that predicates of the form $\underline{A} : \Box bool$ can also be easily detected as they are simply duals of $\underline{E} : \Diamond bool$ which can be detected as shown in Section .

In this paper, we have emphasized conjunctive predicates and not disjunctive predicates. The reason is that disjunctive predicates are quite simple to detect. To detect a disjunctive predicate $\underline{E}$:$\Diamond LP_1 \vee LP_2 \vee \ldots \vee LP_m$, it is sufficient for the process P_i to monitor LP_i. If any of the process finds its local predicate true, the disjunctive predicate is true.

B. Conditions for Weak Conjunctive Predicates

We use LP_i to denote a local predicate in the process P_i, and $LP_i(s)$ to denote that the predicate LP_i is true in the state s. We say that $s \in r[i]$ if s occurs in the sequence $r[i]$.

Our aim is to detect whether $\underline{E}$: $\Diamond(LP_1 \wedge LP_2 \wedge \ldots LP_m)$ holds for a given r. We can assume $m \leq n$ because $LP_i \wedge LP_j$ is just another local predicate if LP_i and LP_j belong to the same process. We now present a theorem which states the necessary and sufficient conditions for a weak conjunctive predicate to hold.

Theorem 6 *$\underline{E}$: $\Diamond(LP_1 \wedge LP_2 \wedge \ldots LP_m)$ is true for a run r* **iff** *for all $1 \le i \le m$ $\exists s_i \in r[i]$ such that LP_i is true in state s_i, and s_i and s_j are incomparable for $i \ne j$. That is, $r \models \underline{E}$: $\Diamond(LP_1 \wedge LP_2 \wedge \ldots LP_m) \Leftrightarrow \exists s_1 \in r[1], s_2 \in r[2], ..., s_m \in r[m]$ such that*
$< \forall i : 1 \le i \le m : LP_i(s_i) > \ \wedge\ < \forall i,j : 1 \le i < j \le m : (s_i \| s_j) >$

Proof: First assume that $\underline{E}$: $\Diamond(LP_1 \wedge LP_2 \wedge \ldots LP_m)$ is true for the run r. By definition, there is a global sequence $g \in linear(r)$ which has a global state, g_* where all local predicates are true. We define $s_i = g_*[i]$ for all i. Clearly, $\forall i : LP_i(s_i) \wedge (s_i \in r[i])$. Now consider any two distinct indices i and j between 1 and m. Since s_i and s_j correspond to the same global state, s_i and s_j must be incomparable by (S2). Therefore, $\exists s_1 \in r[1], s_2 \in r[2], ..., s_m \in r[m] :< \forall i : 1 \le i \le m : LP_i(s_i) > \ \wedge\ < \forall i,j : 1 \le i < j \le m : (s_i \| s_j) >$.

We prove the other direction ($\Leftarrow$) for $m = 2$. The proof for the general case is similar. Assume that there exist states $s_1 \in r[1]$ $s_2 \in r[2]$ such that states s_1 and s_2 are incomparable, and $LP_1(s_1) \wedge LP_2(s_2)$. This implies that there is no message path from s_1 to s_2 or vice-versa. Thus, any message received in or before s_2 could not have been sent after s_1 and any message received in or before s_1 could not have been sent after s_2. Fig. 1. illustrates this. Thus

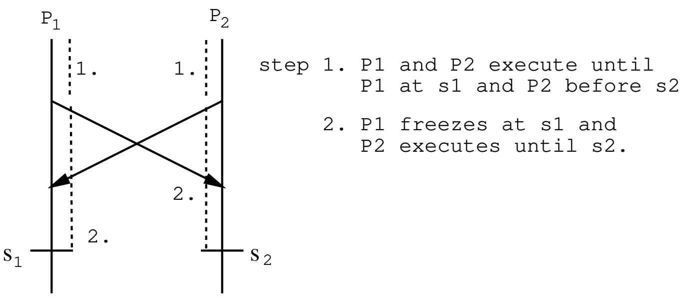

Fig. 1. Incomparable States Producing A Single Global State

it is possible to construct the following execution (global sequence):

1. Let both processes execute consistent with the run r until either P_1 is at s_1 and P_2 is before s_2, or P_2 is at s_2 and P_1 is before s_1. Assume without loss of generality that the former case holds.
2. Freeze P_1 at s_1 and let P_2 execute until it is at s_2. This is possible because there is no message sent after s_1 and received before s_2.

We now have a global state $g_* = (s_1, s_2)$ such that both LP_1 and LP_2 are true in g_*.

■

IV. Detection of Weak Conjunctive Predicates

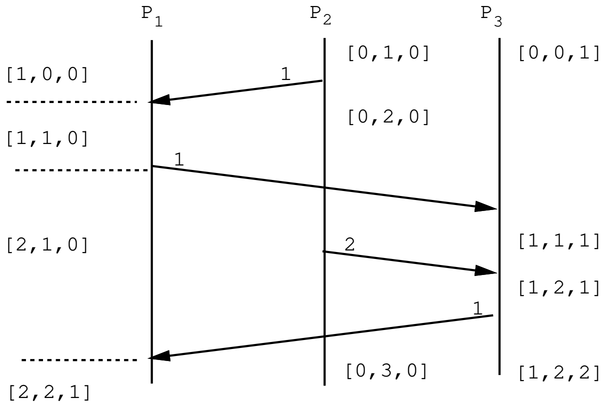

Fig. 2. Examples of lcmvectors

Theorem 6 shows that it is necessary and sufficient to find a set of incomparable states in which local predicates are true to detect a weak conjunctive predicate. In this section, we present a centralized algorithm to do so. Later, we will see how the algorithm can be decentralized. In this algorithm, one process serves as a checker. All other processes involved in WCP are referred to as non-checker processes. These processes, shown in Fig. 3, check for local predicates.

Each non-checker process keeps its own local *lcmvector* (last causal message vector) of timestamps. These timestamp vectors are slight a modification of the virtual time vectors proposed by [6,17]. For the process P_j, lcmvector[i] ($i \ne j$) is the message id of the most recent message from P_i (to anybody) which has a causal relationship to P_j. lcmvector[j] for the process P_j is the next message id that P_j will use. To maintain the lcmvector information, we require every process to include its lcmvector in each program message it sends. Whenever a process receives a program message, it updates its own lcmvector by taking the component-wise maximum of its lcmvector and the one contained in the message. Fig. 2 illustrates this by showing P_1's lcmvector in each interval. Whenever the local predicate of a process becomes true for the *first* time since the most recently sent message (or the beginning of the trace), it generates a debug message containing its local timestamp vector and sends it to the checker process.

One of the reasons that the above algorithm is practical is that a process is not required to send its lcmvector every time the local predicate is detected. A simple observation tells us that the lcmvector need not be sent if there has been no message activity since the last time the lcmvector 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 the lcmvector once after each message is sent irrespective of the number of messages received.

Let $local(s)$ denote that the local predicate is true in state s. We define the predicate $first(s)$ to be true iff the local predicate is true for the first time since the most recently sent message (or the beginning of the trace). We

var
  lcmvector: **array** [1..n] of integer;
    init $\forall i : i \neq id$:lcmvector[i] = 0;
      lcmvector[id] = 1;
    / last causal msg rcvd from process 1 to n */*
  firstflag: boolean **init** true;
  local_pred: Boolean_Expression;
    */*the local pred. to be tested by this process*/*
□ **For** sending **do**
  send (prog, lcmvector, . . .);
  lcmvector[id]++ ; firstflag:=true;
□ **Upon** receive (prog, msg_lcmvector, . . .) **do**
  $\forall i$:lcmvector[i]:=max(lcmvector[i], msg_lcmvector[i]);
□ **Upon** (local_pred = true)$\wedge$ firstflag **do**
  firstflag := false;
  send (dbg, lcmvector) to the checker process;

Fig. 3. Algorithm for weak conjunctive predicates - nonchecker process P_{id}

say $wcp(s_1, s_2, ..., s_m)$ is true if $s_1, s_2, ...s_m$ are the states in different processes making the wcp true (as in Theorem 6).

Theorem 7 $\exists s_1, ..., s_m : wcp(s_1, s_2, ...s_m) \Leftrightarrow$
$< \exists s'_1, .., s'_m : wcp(s'_1, s'_2, ..., s'_m) \wedge \forall i : 1 \leq i \leq m : first(s'_i) >$

Proof: ($\Leftarrow$) is trivially true. We show ($\Rightarrow$). By symmetry it is sufficient to prove the existence of s'_1 such that $wcp(s'_1, s_2, ..., s_m) \wedge first(s'_1)$. We define s'_1 as the first state in the trace of P_1 since the most recently sent message or the beginning of the trace such that $local(s'_1)$ is true. As s_1 exists, we know that s'_1 also exists. By our choice of s'_1, $first(s'_1)$ is true. Our proof obligation is to show that $wcp(s'_1, s_2, ..., s_m)$. It is sufficient to show that $s'_1 \| s_j$ for $2 \leq j \leq m$. For any s_j, $s_1 \not\rightarrow s_j$ and there is no message sent after s'_1 and before s_1; therefore, $s'_1 \not\rightarrow s_j$. Also $s_j \not\rightarrow s'_1$, otherwise $s_j \rightarrow s'_1$ and $s'_1 \rightarrow s_1$ would imply that $s_j \rightarrow s_1$, a contradiction. Therefore, we conclude that $s'_1 \| s_j$ for any $2 \leq j \leq m$.

■

We now analyze the complexity of non-checker processes. The space complexity is given by the array *lcmvector* and is $O(n)$. The main time complexity is involved in detecting the local predicates which is the same as for a sequential debugger. Additional time is required to maintain time vectors. This is $O(n)$ for every receive of a message. In the worst case, one debug message is generated for each program message sent, so the worst case message complexity is $O(m_s)$ where m_s is the number of program messages sent. In addition, program messages have to include time vectors.

We now give the algorithm for the checker process which detects the WCP using the debug messages sent by other processes. The checker process has a separate queue for each process involved in the WCP. Incoming debug messages from processes are enqueued in the appropriate queue. We assume that the checker process gets its message from any process in FIFO order. Note that we do not require FIFO for the underlying computation. Only the detection algorithm needs to implement FIFO property for efficiency purposes. If the underlying communication is not FIFO, the checker process can ensure that it receives messages from non-checker processes in FIFO by using sequence numbers in messages.

The checker process applies the following definition to determine the order between two lcmvectors. For any two lcmvectors, u and v, $u < v$ if and only if $(\forall i : u[i] \leq v[i]) \wedge (\exists j : u[j] < v[j])$. Furthermore, if we know the processes the vectors came from, the comparison between two lcmvectors can be made in constant time. Let $Proc : \mathcal{N}^n \rightarrow \{1, 2, ..., n\}$ map a lcmvector to the process it belongs to. Then, the required computation to check if the lcmvector u is less than the lcmvector v is

$$(u[Proc(u)] \leq v[Proc(u)]) \wedge (u[Proc(v)] < v[Proc(v)]) \quad (P1)$$

Lemma 8 *Let s and t be states in processes P_i and P_j with lcmvectors u and v, respectively. Then, $s \rightarrow t$* **iff** *$u < v$.*

Proof: $(s \rightarrow t) \Rightarrow (u < v)$
If $s \rightarrow t$, then there is a message path from s to t. Therefore, since P_j updates its lcmvector upon receipt of a message and this update is done by taking the component-wise maximum, we know the following holds: $\forall k : u[k] \leq v[k]$. Furthermore, since $v[j]$ is the next message id to be used by P_j, P_i could not have seen this value as $t \not\rightarrow s$. We thereby know that $v[j] > u[j]$. Hence, the following holds: $(\forall k : u[k] \leq v[k]) \wedge (u[j] < v[j])$ Thus, $(s \rightarrow t) \Rightarrow (u < v)$.

We now show that $\neg(s \rightarrow t) \Rightarrow \neg(u < v)$. First, $\neg(s \rightarrow t) \Leftrightarrow (t \rightarrow s) \vee (s \| t)$. If $(t \rightarrow s)$ then $(v < u)$ by the first part of this theorem. If $(s \| t))$ then there is no message path from the state s to the state t or vice-versa. Hence, when P_i is at s and P_j is at t,
(1) $(u[j] < v[j])$ and,
(2) $(v[i] < u[i])$.
Therefore, $(s \| t)) \Rightarrow \neg(u < v)$.

■

Thus, the task of the checker process is reduced to checking ordering between lcmvectors to determine the ordering between states. The following observation is critical for reducing the number of comparisons in the checker process:

Lemma 9 *If the lcmvector at the head of one queue is less than the lcmvector at the head of any other queue, then the smaller lcmvector may be eliminated from further consideration in checking to see if the WCP is satisfied.*

Proof: In order for the WCP to be satisfied, we must find a set of lcmvectors, one from each queue, such that each is incomparable with all the others in the set. If the lcmvector at the head of one queue (q_i) is less than that at the head of another queue (q_j), we know it will be less than any other lcmvectors in q_j because the queues are in increasing order from head to tail. Also any later arrivals into q_j must be greater than that at the head of q_i. Hence, no entry in q_j will ever be incomparable with that at the head of q_i so the head of q_i may be eliminated from further consideration in checking to see if the WCP is satisfied.

■

The algorithm given in Fig. 4 is initiated whenever any new lcmvector is received. If the corresponding queue is non-empty, then it is simply inserted in the queue; otherwise, there exists a possibility that the conjunctive predicate may have become true. The algorithm checks for incomparable lcmvectors by comparing only the heads of queues. Moreover, it compares only those heads of the queues which have not been compared earlier. For this purpose, it uses the variable *changed* which is the set of indices for which the head of the queues have been updated. The *while* loop maintains the invariant:

$(I) \quad \forall i,j \notin changed : \neg empty(q_i) \wedge \neg empty(q_j) \Rightarrow head(q_i) || head(q_j)$

This is done by finding all those elements which are lower than some other elements and including them in *changed*. This means that there can not be two comparable elements in $\{1, 2, ..., m\} - changed$. The loop terminates when *changed* is empty. At that point, if all queues are non-empty, then by the invariant I, we can deduce that all the heads are incomparable. Let there be m queues with at most p elements in any queue. The next theorem deals with the complexity of the above algorithm.

```
var
q1 ... qm: queue of lcmvector;
changed, newchanged: set of {1,2,...,m}
□ Upon recv(elem) from Pk do
    insert(qk, elem);
    if (head(qk) = elem) then begin
        changed := { k };
        while (changed ≠ φ) begin
            newchanged := {};
            for i in changed, and j in {1,2,...,m }do
            if (¬empty(qi) ∧ ¬empty(qj)) then
            begin
                if head(qi) < head(qj) then
                    newchanged:=newchanged ∪ {i};
                if head(qj) < head(qi) then
                    newchanged:=newchanged ∪ {j};
            end; /* if */
            changed := newchanged;
            for i in changed do deletehead(qi);
        end;/* while */
        if ∀i : ¬empty(qi) then found:=true;
    end; /* if */
```

Fig. 4. Algorithm for weak conjunctive predicates - the checker process

Theorem 10 *The above algorithm requires at most $O(m^2p)$ comparisons.*

Proof: Let $comp(k)$ denote the number of comparisons required in the k^{th} iteration of the while loop. Let t denote the total number of iterations of the while loop. Then, the total number of comparisons equals $\sum_{k=1}^{t} comp(k)$. Let $changed(k)$ represent the value of *changed* at the k^{th} iteration. $|changed(k)|$ for $k \geq 2$ represents the number of elements deleted in the $k-1$ iteration of the while loop. From the structure of the for-loops we get that $comp(k) = O(m * |changed(k)|)$. Therefore, the total number of comparisons required are

$\sum_{k=1}^{t} comp(k) = m * \sum_{k=1}^{t} |changed(k)|$

$\leq m * pm = O(m^2p)$

■

The following theorem proves that the complexity of the above problem is at least $\Omega(m^2p)$, thus showing that our algorithm is optimal [8].

Theorem 11 *Any algorithm which determines whether there exists a set of incomparable vectors of size m in m chains of size at most p, makes at least $pm(m-1)/2$ comparisons.*

Proof: We first show it for the case when the size of each queue is exactly one, i.e. $p = 1$. The adversary will give to the algorithm a set in which either zero or exactly one pair of elements are comparable. The adversary also chooses to answer "incomparable" to first $m(m-1)/2 - 1$ questions. Thus, the algorithm cannot determine if the set has a comparable pair unless it asks about all the pairs.

We now show the result for a general p. Let $q_i[k]$ denote the k^{th} element in the queue q_i. The adversary will give the algorithm q_i's with the following characteristic:

$$\forall i, j, k : q_i[k] < q_j[k+1]$$

Thus, the above problem reduces to p instances of the problem which checks if any of the m elements is incomparable. If the algorithm does not completely solve one instance then the adversary chooses that instance to show m queues consistent with all the its answers but different in the final outcome.

■

V. Decentralization of the Detection Algorithm

We now show techniques for decentralizing the above algorithm. From the property (P1), we can deduce that if a set of vectors S forms an anti-chain (that is all pairs of vectors are incomparable), then the following holds:

$$\forall \; distinct \; s,t \in S : s[Proc(s)] > t[Proc(s)] \qquad (P2)$$

We denote this condition by the predicate $inc(S)$. The following theorem shows that the process of checking $inc(S)$ can be decomposed into that of checking it for smaller sets.

Theorem 12 *Let S, T, and U be sets of lcmvectors, such that $S = T \cup U$. Let max_X represent the lcmvector formed by taking componentwise maximum of all vectors in the set X. Then,*

$inc(S)$ iff $inc(T) \wedge inc(U) \wedge$

$(\forall t \in T : max_T[Proc(t)] > max_U[Proc(t)]) \wedge$

$(\forall u \in U : max_U[Proc(u)] > max_T[Proc(u)])$

Proof:($\Rightarrow$) $inc(T)$ and $inc(U)$ are clearly true because $T, U \subseteq S$. We show that

$$(\forall t \in T : max_T[Proc(t)] > max_U[Proc(t)])$$

The other conjunct is proved in a similar fashion.

From $(P2)$, we deduce that $\forall$ *distinct* $s,t \in T : t[Proc(t)] > s[Proc(t)]$. This means that $max_T[Proc(t)] = t[Proc(t)]$ by the definition of max_T. From $(P2)$, we also deduce that $\forall\ u \in U : t[Proc(t)] > u[Proc(t)]$. This means that $t[Proc(t)] > max_U[Proc(t)]$ by the definition of max_U. From the above two assertions we conclude that $(\forall t \in T : max_T[Proc(t)] > max_U[Proc(t)])$

$(\Leftarrow)$ We will show that $(P2)$ holds for S, i.e.
$\forall$ *distinct* $s,t \in S : s[Proc(s)] > t[Proc(s)]$
If both s and t belong either to T and U, then the above is true from $inc(T)$ and $inc(U)$. Let us assume without loss of generality that $t \in T$ and $u \in U$. We need to show that $t[Proc(t)] > u[Proc(t)]$ (the other part is proved similarly). From $inc(T)$ we conclude that, $max_T[Proc(t)] = t[Proc(t)]$. And now from $max_T[Proc(t)] > max_U[Proc(t)]$ we conclude that $t[Proc(t)] > u[Proc(t)]$.

■

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

Example 13 Consider a distributed program with four processes. Let the lcmvectors corresponding to these processes be $S = \{(4,4,6,2),(3,6,4,1),(3,5,7,2),(2,5,4,3)\}$. Now instead of checking whether the entire set consists of incomparable vectors, we divide it into two subsets $T = \{(4,4,6,2),(3,6,4,1)\}$, and $U = \{(3,5,7,2),(2,5,4,3)\}$. We check that each one of them is incomparable. This computation can be done by group checker processes. Group processes send $max_T = (4,6,6,2)$ and $max_U = (3,5,7,3)$ to the higher-level process. This process can check that max_T is strictly greater than max_U in the first two components and max_U is strictly greater than max_T in the last two components. Hence, by Theorem 12, all vectors in the set S are pairwise incomparable.

VI. Implementation: UTDDB

The main application of our results are in debugging and testing of distributed programs. We have incorporated our algorithms in the distributed debugger called UTDDB (University of Texas Distributed Debugger) [14]. The online debugger is able to detect global states or sequences of global states in a distributed computation. UTDDB consists of two types of processes - *coordinator* and *monitor* type. There exists only one coordinator process, but the number of monitor processes is the same as the number of *application* processes in the underlying distributed computation.

The coordinator process serves as the checker process for WCP as well as the user-interface of UTDDB to the programmer. It accepts input from the programmer such as distributed predicates to be detected. It also reports to the programmer if the predicate is detected.

Monitor process are hidden from the programmer. Each of the monitor processes, detects local predicates defined within the domain of the application process it is monitoring. This is done by single stepping the program. After each step, the monitor examines the address space of the application process to check if any of the simple predicates in its list are *true*. It is also responsible to implement algorithms described as a non-checker process in Section . In particular, it maintains the vector clock mechanism.

In a distributed debugger, the delays between occurrence of a predicate, its detection and halting of the program may be substantial. Thus, when the program is finally halted, it may no longer be in a state the programmer is interested in. Therefore, for the weak conjunctive predicate, UTDDB gives the programmer the option of rolling back the distributed computation to a consistent global state where the predicate is true. The coordinator uses the set of timestamps that detected the WCP predicate to calculate this global state which it then sends to all the monitors. As the application processes execute, they record incoming events to a file. So, when a monitor receives a message telling it to roll back an application process, the monitor restarts the application process and replays the recorded events until the process reaches a local state that is part of the global state where the weak conjunctive predicate is true. Such a restart assumes that the only non-determinism in the program is due to reordering of messages.

Our algorithms are also used in a trace analyzer (another part of UTDDB) for distributed programs [4]. Our analyzer monitors a distributed program and gathers enough information to form a distributed run as described in Section II. This approach reduces the probe effect that the distributed program may experience if the detection was carried out while the program was in execution. The user can then ask UTDDB whether any predicate expressed in a subset of the logic described in this paper ever became true. We are currently extending these algorithms for detection of sequences of global predicates [1,9,25], and relational global predicates [24].

VII. 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.

Our experience with these algorithms has been extremely encouraging. In the current implementation, the main overhead is in the local monitor process for checking local predicates. By providing special hardware support even this overhead can be reduced. For example, most architectures provide special hardware support such as breakpoint traps if certain location is accessed. This feature can be used to make detection of local predicates of the form (program at line x) very efficient.

We believe that algorithms presented in this paper should be part of every distributed debugger because they incur low overhead, and are quite useful in identifying errors in the program.

Acknowledgements

We would like to thank Bryan Chin, Mohamed Gouda, Greg Hoagland, Jay Misra, William Myre, Don Pazel, and Alex Tomlinson for their comments and observations which have enabled us to strengthen this work. We would also like to thank Bryan Chin for implementing offline versions of our algorithms and Greg Hoagland for incorporating our algorithms in UTDDB. We would also like to thank anonymous referees for their meticulous review of an earlier version of the paper.

References

[1] P. Bates, "Distributed Debugging Tools for Heterogeneous Distributed Systems", *Proceedings of the 8-th International Conference on Distributed Computing Systems*, San Jose, California, pp. 308 – 315, June 1988.

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

[3] K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems", *ACM Transactions on Computer Systems*, Vol. 3, No. 1, pp. 63 – 75, February 1985.

[4] B. Chin, "An Offline Debugger for Distributed Programs", M.S. Thesis, Electrical and Computer Engineering, University of Texas at Austin, December 1991.

[5] R. Cooper and K. Marzullo, "Consistent Detection of Global Predicates", *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, California, pp. 163 – 173, May 1991.

[6] C. Fidge, "Partial Orders for Parallel Debugging", *Proc. of the ACM Workshop on Parallel and Distributed Debugging*, Madison, Wisconsin, pp. 130 – 140, May 1988.

[7] J. Fowler and W. Zwaenepoel, "Causal Distributed Breakpoints", *Proc. of the 10-th Intl. Conference on Distributed Computing Systems*, Paris, France, pp. 134 – 141, May 1990.

[8] V. K. Garg, "Some Optimal Algorithms for Decomposed Partially Ordered Sets," *Information Processing Letters* 44, November 1992, pp 39-43.

[9] V. K. Garg and M.T. Raghunath, "Concurrent Regular Expressions and their Relationship to Petri Net Languages," *Theoretical Computer Science* 96 (1992) pp 285-304.

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

[11] D. Haban and W. Weigel, "Global events and global breakpoints in distributed systems", *Proc. of the 21-st Intl. Conf. on System Sciences*, Vol. 2, Jan 1988, pp 166 – 175. 1990, pp. 134 – 141.

[12] 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, California, May 1993.

[13] J.-M. Helary, N. Plouzeau, and M. Raynal, "Computing Particular Snapshots in Distributed Systems", *Proceedings of the 9-th annual International Phoenix Conference on Computers and Communications*, Scottsdale Ariz., March 21 – 23, 1990, pp. 116 – 123.

[14] G. Hoagland, "A Debugger for Distributed Programs", M.S. Thesis, Electrical and Computer Engineering, University of Texas at Austin, August 1992.

[15] H. Korth and A. Silberschatz, *Database System Concepts*, McGraw-Hill, 1986.

[16] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, Vol. 21, No. 7, July, 1978, pp. 558 – 565.

[17] F. Mattern, "Virtual time and global states of distributed systems", *Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms*, Elsevier Science Publishers B. V., 1989, pp. 215–226.

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

[19] B. P. Miller and J. Choi, "Breakpoints and Halting in Distributed Programs", *Proceedings of the 8-th International Conference on Distributed Computing Systems*, San Jose, california, June 1988, pp. 316–323.

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

[21] A.P. Sistla and E.M. Clarke, "The Complexity of Propositional Linear Temporal Logic", *J. ACM*, 1985, Vol. 32, No. 3, pp. 733 – 749.

[22] M. Spezialetti and P. Kearns, "Efficient Distributed Snapshots", *Proceedings of the 6-th International Conference on Distributed Computing Systems*, 1986, pp. 382 – 388.

[23] M. Spezialetti and P. Kearns, "A General Approach to Recognizing Event Occurrences in Distributed Computations", *Proc. of the 9-th Interl Conference on Distributed Computing Systems*, 1988, pp. 300 – 307.

[24] A.I. Tomlinson and V.K.Garg, "Detecting Relational Global Predicates in Distributed Systems," Proc. *3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, San Diego, California, May 1993.

[25] B. Waldecker, "Detection of Unstable Predicates in Debugging Distributed Programs", *PhD. Dissertation*, Dept. of Electrical and Computer Engineering, University of Texas at Austin, May 1991.

Vijay K. Garg (S'84-M'89) received his Bachelor of Technology degree in computer engineering from the Indian Institute of Technology, Kanpur, in 1984. He continued his education at the University of California, Berkeley where he received his MS in 1985 and Ph.D. in 1988 in Electrical Engineering and Computer Science. He is currently an assistant professor in the Department of Electrical and Computer Engineering at the University of Texas, Austin. He has authored or co-authored more than 50 research articles. His research interests are in the areas of distributed systems and supervisory control of discrete event systems. He has served as a program committee member of the IEEE International Conference on Distributed Computing Systems and as an organizer of minisymposium on Discrete Event Systems in the SIAM Conference on Control and Applications. Prof. Garg is a recipient of the 1992 TRW faculty assistantship award.

Brian Waldecker received the B.S.E.E and B.A. in computer science degrees from Rice University in Houston, TX, in 1986, and the M.S. and Ph.D. degrees in electrical and computer engineering from the University of Texas at Austin, in 1988 and 1991, respectively. He is currently with the Austin Systems Center of Schlumberger Well Services where he works on software for oilfield services. His interests include distributed computing systems and distributed program behavior. He is a member of the IEEE Computer Society, the Association for Computing Machinery, and the Society of Petroleum Engineers.