Detecting Conjunctive Channel Predicates in a Distributed Programming Environment

Vijay. K. Garg   Craig. Chase   J. Roger Mitchell   Richard Kilgore

TR   ECE-PDS-94-02   June   1994

Parallel & Distributed Systems group
Department of Electrical & Computer Engineering
University of Texas at Austin
Austin, Texas 78712

Detecting Conjunctive Channel Predicates in a Distributed Programming Environment

Vijay K. Garg* Craig Chase† J. Roger Mitchell‡ Richard Kilgore

Parallel and Distributed Systems Laboratory
email: pdslab@ece.utexas.edu
Electrical and Computer Engineering Department
The University of Texas at Austin,
Austin, TX 78712

Abstract

This paper discusses efficient detection of global predicates in a distributed program. Previous work in efficient detection of global predicates was restricted to predicates that could be specified as a boolean formula of local predicates. Many properties in distributed systems, however, use the state of channels. In this paper, we introduce the concept of a channel predicate and provide an efficient algorithm to detect any boolean formula of local and channel predicates. We define a property called monotonicity for channel predicates. Monotonicity is crucial for efficient detection of global predicates. We show that many problems studied earlier such as detection of termination and computation of global virtual time are special cases of the problem considered in this paper. 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 programming environment which runs on a network of IBM RS/6000 Workstations under AIX with the PVM environment.

1 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, that is, a condition that depends on the state of the entire system, is a fundamental problem in distributed computing. This problem arises in many contexts such as designing, testing and debugging of distributed programs.

Previous work has described algorithms for detecting stable and unstable global predicates [2, 3, 6, 8, 9, 11, 13, 14, 15, 16, 20]. See [1, 18] for surveys of stable and unstable predicate detection. Stable predicates are those that never become false once they are true. The often cited examples of stable predicates are deadlock and termination. For example, a system that has terminated remains in this state. Chandy and Lamport's method [2] for detecting a global predicate involves periodically taking a global snapshot of the state of the system.

*supported in part by the NSF Grant CCR-9110605, a TRW faculty assistantship award, a General Motors Fellowship, and an IBM grant

†supported in part by the Texas Instruments/Jack Kilby Faculty Fellowship

‡supported in part by an MCD fellowship

Unlike stable predicates, unstable predicates may alternate between true and false values. Cooper and Marzullo [3] presented a method for detecting general predicates which included unstable predicates. Their approach, though, requires $O(k^n)$ time where k is the maximum number of events a monitored process has executed and n is the number of processes. In particular, their algorithm checks *all* possible global states, whereas ours does not.

Manabe and Imase [15] presented a method for detecting predicates, including channel predicates, using a replay approach. This approach requires two identical runs and restricts channel predicates to those detectable by a process. Our approach requires only a single run. Furthermore, our channel predicates are more general because they also include predicates that cannot be detected by a single process.

Garg and Waldecker [8] presented a method for detecting weak unstable conjunctive predicates where local processes check for their own predicate and send a message to a global predicate checker whenever the predicate becomes true between application messages.

Our detection of global predicates extends the algorithms used in the detection of weak unstable predicates [8] to include the state of communication channels. A channel is a uni-directional connection between any two processes through which messages can be passed. A general mechanism for the detection of channel predicates as part of global predicates is an important characteristic for distributed debuggers. Furthermore, many classical problems, such as distributed termination, and bounding of global virtual time can be detected by our algorithm.

The key to making our algorithm efficient is to restrict the channel predicates to a class which we call *monotonic*. The basis of this class are send-monotonic predicates and receive-monotonic predicates. A send-monotonic predicate cannot be made true by sending more messages along the same channel. For example, consider the condition "The channel is empty." If this condition is false, that is, more messages have been sent than received, it cannot be made true by sending still more messages. A receive-monotonic predicate is analogous. When a receive-monotonic predicate is false, it can not be made true by only receiving more messages. The generalization of these two classes of predicates we call *dynamically monotonic* predicates. A dynamically monotonic predicate can behave in some states as a send-monotonic predicate and in others as a receive-monotonic predicate. For example, consider the predicate, "The channel contains exactly 5 messages". When the channel contains less than 5 messages, this predicate is receive monotonic. Receiving more messages will not make the predicate true. If there are more than 5 messages in the channel, then the predicate is send-monotonic, and sending more messages can not make the predicate true.

We show that dynamic monotonicity is an important key to efficient detection of channel predicates. In any global state in which the predicate is false, we can be certain of at least one process which must make further progress before the channel predicate can become true. Monotonicity allows us to guarantee that progress by the other processes can not make the predicate true. Furthermore, we also show that the first global state satisfying a GCP can only be well defined when monotonic channel predicates are used.

For distributed debugging, it is important to identify situations that may not have occurred in the current execution, but because of different clock speeds, could occur in subsequent executions. Our algorithm can detect these types of situations by testing for the occurrence of a global predicate among all *potentially* concurrent states.

In spite of the generality of our technique, the algorithms are efficient enough to be included in a distributed debugger. The time complexity is $O(m^2n + mn^2)$ while the space complexity is $O(mn^2)$ where m is the maximum number of application messages generated by a process and n is the number of processes. We have implemented our algorithm for global predicate detection in a distributed programming environment. Our initial implementation runs on the Parallel Virtual Machine (PVM) infrastructure on a network of IBM RS/6000 workstations.

The next section will present the notation, definitions of predicates, and the model, which are necessary in understanding the method of detecting conjunctive channel predicates. Sections 3 and 4 discuss the checker and non-checker processes of our model and their operation. Section 5 demonstrates correctness of the algorithm and Section 6 presents our experimental results.

2 Our Model

This section presents the concepts and notation of distributed runs, and global, local and channel predicates.

2.1 Distributed Run

We assume a loosely-coupled message-passing system without any shared memory or a global clock. A distributed program consists 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*, with implied actions between these states. That is, the process P_i generates the trace $r[i] = \alpha_{i,0}\alpha_{i,1}\ldots\alpha_{i,k}$, where α_i's are the local states, and where k is the maximum number of distinct states in a single process. There are three kinds of events that can occur between these states — an internal event, sending of a message and reception of a message. Finally, the state of a process is defined by the value of all its variables including its program counter.

We assume that no messages are lost, altered or spuriously introduced. We do not make any assumptions about a FIFO nature of the channels. We also define a happened-before relation, '$\rightarrow$,' between states similar to that of Lamport's happened-before relation between events [13]. The happened-before relation can be formally stated as:

$\alpha \rightarrow \beta$ iff:

1. $\alpha \prec \beta$ where $\prec$ means occurred before in the same process, or
2. $\alpha \rightsquigarrow \beta$ where $\rightsquigarrow$ means that the action following α is a send of a message and the action preceding β is a receive of that message, or
3. $\exists \gamma : \alpha \rightarrow \gamma \wedge \gamma \rightarrow \beta$.

The intuition behind the happened-before relation is that there is a *message path* between two states for which this relation holds.

2.2 Global Sequence

A run defines a partial order using happened-before on the set of actions and states. In general, there are many total orders, or linearizations, that are consistent with this partial order. A *global sequence* corresponds to a view of the run which could be obtained given the existence of a global clock. A global sequence, g, is a finite sequence denoted as $g = g_0 g_1 \ldots g_k$, where g_i is a global state for $0 \leq i \leq k$. Each g_i consists of a vector of process and channel states. This definition is the same as that of Chandy and Lamport[2]. In addition to global states, we use *local snapshots* at a process to mean the local state and the history of activity of all channels incident to that process.

2.3 Global predicates

The concept of concurrency is required in the detection of global predicates. Two states for which the happened-before relation does not hold in either direction are said to be concurrent. That is, there exists a run with local clocks running at particular speeds where two concurrent states can be made to occur at the same physical time. The symbol, $\|$, is used to represent concurrency. The relationship can be formally stated as:

$$\alpha \parallel \beta \Leftrightarrow (\alpha \not\to \beta \ \wedge \ \beta \not\to \alpha)$$

Given a set of n states, if this condition holds for all pairwise combinations of the states, then this set forms a *consistent cut*. The desire is to detect conditions at processes and within channels distributed across a system that could *potentially* occur at the same time. The method of doing this is by detecting concurrency of the process and channel states for which all the local conditions, or predicates, are true. The idea of local and channel predicates are now given in more detail.

2.3.1 Local Predicates

A local predicate is defined as any boolean formula on a local process. For any process, represented by P_i, a local predicate is written as l_i. $l_i(\alpha)$ is used to represent the predicate being true in a particular state, α, of P_i. A process can obviously detect a local predicate on its own.

2.3.2 Channel Predicates

A channel predicate is any boolean function of the state of the channel. The channel state is defined as the set difference of the send events and the receive events on that channel. Since a channel is defined as the uni-directional connection between two processes, one process performs all send events and the other all receive events. We therefore define a channel predicate in terms of all send events from one process and all receive events at another. The notation for the accumulation of send and receive events is defined first:

α, β:	states at different processes, P_i and P_j, or $\alpha \in r[i]$ and $\beta \in r[j]$.
$\alpha.Sent[j]$:	sequence of all messages sent at or before state α from i to j.
$\beta.Rcvd[i]$:	sequence of all messages received at or before state β from i to j.

A channel predicate can then be written as:

$$c(\alpha.Sent[j] - \beta.Rcvd[i])$$

or in short notation as:

$$c(\alpha, \beta) \equiv c(\alpha.Sent[j] - \beta.Rcvd[i])$$

In this paper, we will use the symbol S to represent an arbitrary sequence of sends events from a process and the symbol R to represent an arbitrary sequence of receive events. It should be noted that channels have no memory. Hence, any channel state that can be constructed by a combination of both send events and receive events can also be produced by some other sequence of just send events.

For efficiency reasons the channel predicates must be at least dynamically monotonic. The requirement for dynamic monotonicity can be stated formally as:

Definition 2.1 *A channel predicate, c(S), is said to be* dynamically monotonic *iff:*

$$\forall S :: \neg c(S) \Rightarrow (\forall S' :: \neg c(S \cup S')) \vee (\forall R :: \neg c(S - R))$$

That is, given any channel state, S, in which the predicate is false, then either sending more messages is guaranteed to leave the predicate false, or receiving more messages is guaranteed to leave the predicate false. We assume that when the channel predicate is evaluated in some state S, it is also known which of these two cases applies. To model this assumption, we define monotonic predicates to be 3-valued functions. The predicate can evaluate to:

1. T — The channel predicate is true for the current channel state.
2. F_s — The channel predicate is false for the current channel state. Furthermore, the predicate will remain false in the presence of an arbitrary set of additional messages sent on the channel in the absence of receives.
3. F_r — The channel predicate is false for the current channel state. Furthermore, the predicate will remain false in the presence of an arbitrary set of messages received from the channel in the absence of sends.

Example 1. *Detection of empty channels:* $c(S) \equiv (S = \emptyset)$: It is obvious that if this predicate is not currently true, sending more messages will not make it become true. This send-monotonic predicate can be used in termination detection, as demonstrated in Section 6.

Example 2. *Detection of at least k messages in a channel:* $c(S) \equiv$ (number of messages in $S \geq k$): This is similar to the detection of empty channels. This receive-monotonic predicate can be used in buffer overflow detection.

Example 3. *Detection of exactly k messages in a channel:* $c(S) \equiv$ (length of $S = k$): In any state where there are more than k messages in the channel, the predicate cannot be made true by sending more messages. In any state when there are less than k messages in a channel, the predicate can not be made true by receiving more messages. The only other possible state is when the channel has exactly k messages in it, and in this state the predicate is true.

2.3.3 Generalized conjunctive predicate

We call a predicate detected by our algorithm a generalized conjunctive predicate (GCP). A GCP is true for a given run if and only if there exists a global sequence for that run in which all conjuncts are true in some global state. We define the GCP to be conjunctive because the capability to detect any conjunctive predicate is sufficient to support the detection of any global predicate.

$$GCP = (l_1 \wedge l_2 \wedge ...l_n \wedge c_1 \wedge c_2 \wedge ...c_e)$$

Lemma 2.2 *Let p be any global predicate constructed from local and channel predicates using boolean connectives. Then, p can be detected using an algorithm that can detect q where q is a GCP.*

Proof: We first write p in disjunctive normal form. Thus, $p = (q_1 \vee \ldots \vee q_k)$ where each q_i is a pure conjunction of local predicates. Next, we observe that since p is a disjunction, the detection of any q_i in a global state will make p true. Finally, since each predicate q_i is a conjunctive predicate, the detection of p can be broken down into the detection of any one of k conjunctive predicates. □

The following theorem is useful in detecting a GCP.

Theorem 2.3 *$(l_1 \wedge l_2 \wedge ...l_n \wedge c_1 \wedge c_2 \wedge ...c_e)$ true in a run $r \Leftrightarrow$*

$$\exists \alpha_1, \alpha_2, ...\alpha_n : \forall i, j\ (\alpha_i \parallel \alpha_j\ \wedge\ l_i(\alpha_i)\ \wedge\ c(\alpha_i.Sent[j] - \alpha_j.Rcvd[i]))$$

Proof: Follows directly from the definition of a conjunctive predicate (concurrency) and the definition of channel predicates. □

The next theorem describes the structure of cuts satisfying a GCP. Let $\mathcal{C}$ be the set of all global cuts that satisfy a GCP with monotone channel predicates. For two cuts $C, D \in \mathcal{C}$, we say that $C \leq D$ iff $\forall i : C[i] \preceq D[i]$ where $C[i]$ is the state from P_i in C and $\preceq$ means $\prec$ or $=$. We show that the concept of *first* cut that satisfies a GCP is well-defined. In other words, if two global cuts satisfy a GCP, then their greatest lower bound also satisfies that GCP.

Theorem 2.4 *Let a GCP be such that all of its channel predicates are monotone. Let $(\mathcal{C}, \leq)$ be the set of all global cuts in which the GCP is true. If $C, D \in \mathcal{C}$, then their greatest lower bound is also in $\mathcal{C}$.*
Proof: Let E be defined as $E[i] = min(C[i], D[i])$ and $chanp(E[i], E[j])$ denote the value of the channel predicate between processes P_i and P_j at states $E[i]$ and $E[j]$. We show that $E \in \mathcal{C}$, that is, E also satisfies the GCP. There are three properties that E must satisfy: all local predicates must be true, all states in E must be concurrent, and all channel predicates must be true.

1. Since $E[i]$ is either $C[i]$ or $D[i]$, and both $l_i(C[i])$ and $l_i(D[i])$ hold, it follows that $\forall i : l_i(E[i])$.

2. Let
$$I = \{i | E[i] = C[i]\} \text{ and : } \ J = \{i | E[i] = D[i]\}.$$
It is clear that $\forall i, j \in I : E[i] || E[j]$ and that $\forall i, j \in J : E[i] || E[j]$. We now show that $\forall i \in I, j \in J : E[i] || E[j]$. Since $E[i] = C[i]$, $C[i] || C[j]$ and $D[j] \preceq C[j]$, it follows that $E[j] \not\rightarrow E[i]$. Similarly, $E[j] = D[j]$, $D[i] || D[j]$ and $C[i] \preceq D[i]$ implies that $E[i] \not\rightarrow E[j]$. Therefore, we have shown that E is a consistent cut.

3. We now show that E also satisfies channel predicates. By symmetry, it is sufficient to show that $\forall i \in I, j \in J : chanp(E[i], E[j])$. Assume for contradiction, that $chanp(E[i], E[j])$ is false. By monotonicity of channel predicates, there are two cases:

   Case 1 $chanp(E[i], E[j]) = F_s$ — Since $E[i] \preceq D[i]$ and process i can only send on this channel, this would imply that $chanp(D[i], E[j])$ is false, hence $chanp(D[i], D[j])$ is false, a contradiction.

   Case 2 $chanp(E[i], E[j]) = F_r$. — Since $E[j] \preceq C[j]$, this would imply that $chanp(C[i], C[j])$ is false, a contradiction.

   Therefore, all channel predicates must also be true in E.

Therefore, the GCP is satisfied by the cut E. □

The above theorem does not hold for arbitrary channel predicates as shown by the next example.

Example 1 Consider the distributed computation shown in Figure 1. Consider the channel predicate — "There are an odd number of messages in the channel." Note that this channel predicate is not dynamically monotonic. Assume that the local predicates are true only at points $C[1]$ and $D[1]$ for P_1, and $C[2]$ and $D[2]$ for P_2. It is easily verified that the GCP is true in the cut C and D but not in their greatest lower bound.

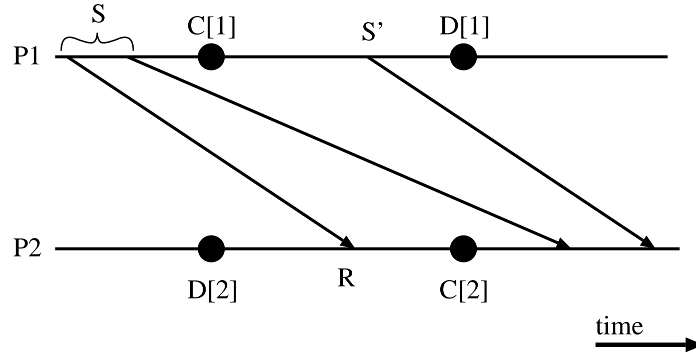

Figure 1: An example to show that the set of cuts satisfying a GCP is not a lattice.

We now show that the first cut satisfying a GCP is always well defined only if channel predicates are restricted to be monotonic. We restrict our consideration to those GCPs which can possibly be true for at least one run of some program.

Theorem 2.5 *The first cut that satisfies GCP is always well defined only if all channel predicates in the GCP are restricted to dynamically monotonic channel predicates.*

Proof: The proof is by contrary example. Given any GCP that includes at least one non-monotonic channel predicate, we can construct a program for which there is no unique first cut satisfying that GCP.

Figure 1 illustrates the situation we wish to construct. Without loss of generality, let P_1 and P_2 be two processes from the GCP such that a non-monotonic channel predicate is used for the channel from P_1 to P_2. All other processes interact so as to make the remaining channel predicates true and then these processes become idle in a consistent cut with local predicates true. Up to this point, there has been no activity on the channel from P_1 to P_2.

Since the channel predicate from P_1 to P_2 is non-monotonic, there exists a channel state for which the channel predicate is false, but can be made true both by sending and by receiving messages. Let S be a set of message sends so that the channel enters this state. The program is constructed such that P_1 performs S on the channel prior to state *C[1]*. In Example 1, the sequence S consists of two arbitrary messages. The local predicate on P_1 then becomes true for the first time in state *C[1]*. The local predicate on P_2 becomes true for the first time in state *D[2]*. Since P_2 has not received any messages from P_1 by state D[2], the channel predicate is not true along the cut defined by *C[1]* and *D[2]*.

Let S' be the set of additional messages that can be sent so that the predicate becomes true. The process P_1 sends these messages between states *C[1]* and *D[1]*. In addition, let R be the set of messages that can be received so that the predicate can be made true. The process P_2 receives R between states *D[2]* and *C[2]*. Note that both cuts C and D are consistent cuts. Furthermore, all local and channel predicates are true on these cuts.

It is clear that $C \not\leq D$ and $D \not\leq C$. Since the local predicates on P_1 and P_2 were not true at any earlier point in this program, there is no cut which is a lower bound of both C and D and that

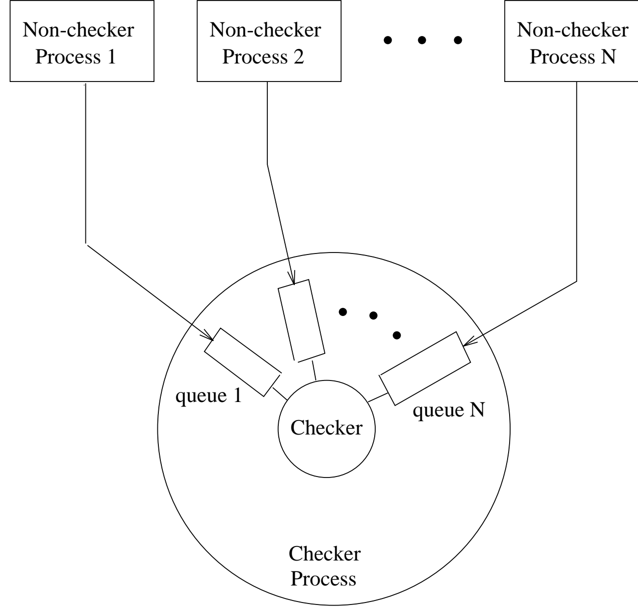

Figure 2: The checker/non-checker GCP detector.

satisfies the GCP.

Therefore, the first cut to satisfy this GCP is not well defined for this program. □

3 GCP Algorithm: The Non-Checker Processes

The method of detection of the GCP is divided among checker and non-checker processes. The non-checker processes are used in the computation and have local predicates and channels with predicates. The checker process is the process that determines if these predicates are true in the same global state. Figure 2 shows the general idea with checker process queues to collect detection messages from the non-checker processes. As Theorem 1 suggests, detection of a generalized conjunctive predicate can be achieved by detecting whether or not local states and states of channels in which predicates are true are all concurrent. To do this requires that processes provide information as to when local predicates are true. If the local processes also provide the send and receive sequences for all channels, then channel predicates can be checked. If all predicates are true concurrently, then the GCP is true.

To perform this concurrency detection, our algorithm makes use of *lcmvectors* [7]. An *lcmvector*, or last causal message vector, operates similar to Mattern's[14] and Fidge's[5] vector clocks except that the vector is only incremented when a process sends a program message. The lcmvector provides the property:

> $\alpha \rightarrow \beta$ iff $\alpha.u < \beta.v$, where α and β are states in processes P_i and P_j $(i \neq j)$ and u and v are the respective lcmvectors at these states

The lcmvector contains one component for each process that is involved in the GCP. The lcmvector is attached to all program messages sent by each process. Program messages are part of the underlying computation and are not part of the detection algorithm.

The non-checker processes monitor local predicates. These processes also maintain information about the send and receive channel history for all channels incident to them, that is, connections

to all processes for which they can send or receive messages. The non-checker processes send a message to the checker process whenever the local predicate becomes true for the first time since the last program message was sent or received. This message is called a local snapshot and is of the form:

$$(lcmvector, incsend, increcv)$$

where *lcmvector* is the current lcmvector while *incsend* and *increcv* are the list of messages sent to and received from other non-checker processes since the last message for predicate detection was sent. An algorithm for this process is given in Figure 3.

```
var
incsend, increcv: array of messages;
lcmvector: array [1..n] of integer;

initially ∀j : j ≠ i :lcmvector[j] = 0;
        lcmvector[i] = 1;
        firsflag = true;
        incsend = increcv = ∅;

for sending m do
        send (lcmvector, m);
        lcmvector[i]++ ;
        firstflag:=true;
        incsend:= incsend ⊕ m; /* concatenate */

upon receive (msg_lcmvector, m) do
        for (all j) lcmvector[j]:=max(lcmvector[j], msg_lcmvector[j]);
        firstflag:=true;
        increcv:= increcv ⊕ m; /* concatenate */

upon (local_pred = true ∧ firstflag) do
        firstflag := false;
        send (lcmvector, incsend, increcv) to the checker process;
        incsend:=increcv:=∅;
```

Figure 3: Non-checker process algorithm for P_i

4 GCP Algorithm: The Checker Process

The checker process is responsible for searching for a consistent cut that satisfies the GCP. Its pursuit of this cut can be most easily described as considering a sequence of candidate cuts. If the candidate cut either is not a consistent cut, or does not satisfy some term of the GCP, the checker can efficiently eliminate one of the states along the cut. The eliminated state can never be part of a consistent cut that satisfies the GCP. The checker can then advance the cut by considering the successor to one of the eliminated states on the cut. If the checker finds a cut for which no state can be eliminated, then that cut satisfies the GCP and the detection algorithm halts. The algorithm for the checker process is shown in Figure 4.

4.1 Data Structures

The checker receives local snapshots from the other processes in the system. These messages are used by the checker to create and maintain data structures that describe the global state of the system for the current cut. The data structures are divided into three categories: queues of incoming

```
S[1..n,1..n], R[1..n,1..n] : sequence of message;
cp[1..n,1..n] : {X, F, T};
cut : array[1..n] of struct {
    v : vector of integer;
    color : {red, green};
    incsend, increcv : sequence of messages }
initially
    cut[i].v = 0; cut[i].color = red; S[i,j] = ∅; R[i,j] = ∅;

repeat
    /* advance the cut */
    while (∃ i : (cut[i].color = red) ∧ (q[i] ≠ ∅))
        cut[i] := receive(q[i]);
        paint-state(i);
        update-channels(i);
    endwhile

    /* evaluate a channel predicate */
    if (∃ i,j : cp[i,j] = X ∧ cut[i].color = green ∧ cut[j].color = green) then
        cp[i,j] := chanp(S[i,j]);
        if (cp[i,j] = F_s) cut[j].color := red;
        else if (cp[i,j] = F_r) cut[i].color := red;
    endif
until (∀ i : cut[i].color = green) ∧ (∀ i,j : cp[i,j] = T)
detect := true;
```

Figure 4: GCP Detection Algorithm, Checker Process

messages, those data structures that describe the state of the processes, and those data structures that include information describing the state of the channels.

4.1.1 Incoming Message Queues

The checker relies on being able to selectively receive a message from a specific process. For example, at some phase in the algorithm the checker may ask to receive a message sent specifically by process i. Furthermore, we require that messages from an individual process be received in FIFO order. These capabilities are provided by most modern message passing systems, such as PVM [10] and MPI [21]. If the message passing system did not provide this support, it can be easily constructed using a set of FIFO queues.

We abstract the message passing system as a set of n FIFO queues, one for each process. We use the notation *q[1..n]* to label these queues in our algorithm. We abstract non-blocking message reception as the ability to compare a queue to the empty set.

4.1.2 Per-Process Data

The checker maintains information describing one state from each process P_i. The collection of this information is organized into a vector:

cut : array[1..n] of struct process_data

The *process_data* structure consists of a local snapshot (see Section 3) plus the following item:

- *color : {red, green}* — The color of a state is either red or green and indicates whether the state has been eliminated in the current cut. A state is green only if it is concurrent with all other green states. A state is red only if it cannot be part of a consistent cut that satisfies the GCP.

4.1.3 Per-Channel Data

The checker maintains three data structures for each channel:

- *S[1..n, 1..n] : sequence of messages* — The pending-send list (or "S" list). This list is an ordered list of messages. The list contains all those messages that have been sent on the channel, but not yet received according to the current cut.

- *R[1..n, 1..n] : sequence of messages* — The pending-receive list (or "R" list). The list contains each message that has been received from the channel, but not yet sent according to the current cut. Since the current cut is not necessarily consistent, states along the cut may be causally related, and hence it is possible for one state on the cut to be after a message has been received, and yet have another state on the cut from before that message was sent. If all states are part of a consistent cut, then every R list is empty.

- *cp[1..n, 1..n] :* $\{X, F_s, F_r, T\}$ — The CP-state flag. When a channel predicate is evaluated, its value is written into the CP-state flag. The value of a channel predicate cannot change unless there is activity along the channel. Hence, the checker can avoid unnecessarily recomputing channel predicates by recording which predicates have remained true or false since the last time the predicate was evaluated. If the CP-state flag has any value other than X, then that value must be the value of the channel predicate for the current cut. The CP-state flag can take the value X at any time. The value X indicates the current value of the channel predicate is unknown.

4.2 Checker Algorithm

There are two main activities for the checker inside the repeat loop shown in Figure 4. The first activity advances the current cut. The second activity evaluates channel predicates for channels between two concurrent states in the cut. Advancing the current cut is given a higher priority than evaluating channel predicates. Channel predicates are only evaluated either when the current cut is a consistent cut satisfying all local predicates, or when the checker cannot advance the current cut because sufficient messages have not yet arrived from the processes. The checker continues executing the two activities until the GCP is detected. As an obvious extension, if some process has terminated and none of the states received from that process satisfy the GCP, the checker can abort the detection algorithm.

4.2.1 Advancing the Cut

The aim of this activity is to find a new candidate cut. However, we can advance the cut only if we have eliminated at least one state along the current cut and if a message can be received from the corresponding process. The data structures for the processes and channels are updated to reflect the new cut. This is done by the procedures *paint-state* and *update-channels* respectively.

We first consider the procedure *paint-state*. This procedure is shown in Figure 5. The parameter i is the index of the process from which a local snapshot was most recently received. The color of *cut[i]* is temporarily set to green. It may be necessary to change some green states to red in order

to preserve the property that all green states are mutually concurrent. Hence, we must compare the vector clock of *cut[i]* to each of the other green states. Whenever the states are comparable, the smaller of the two is painted red. Observe that once we paint *cut[i]* red, we can stop attempting to paint other states red. If this state is smaller than any green state, then by transitivity it cannot be larger than any of the other green states which are known to be mutually concurrent.

```
paint-state(i)

    cut[i].color := green;
    for (j : cut[j].color = green) do
        if (cut[i].v < cut[j].v) then
            cut[i].color := red;
            return
        else if (cut[j].v < cut[i].v) cut[j].color := red;
        endif
    endfor
```

Figure 5: Procedure paint-state

We now consider the procedure *update-channels*. This procedure is shown in Figure 6. As with *paint-state*, the parameter i is the index of the process from which a local snapshot was most recently received. The checker updates the value of the CP-state flags according to the activity in *cut[i].incsend* and *cut[i].increcv*. In the worst case, each message sent or received causes the CP-state flag to be reset to X. The checker will never change the CP-state flag to any value other than X while advancing the cut. As an optimization, the checker can take advantage of monotonicity when updating the channel-state vector. If a channel predicate is false along the current cut, and that predicate is currently send-monotonic, then it will remain false when more messages are sent. There will be no need to evaluate the predicate until at least one message receive occurs on the channel. There is a similar optimization for states when the predicate is receive-monotonic.

The incremental send and receive histories from the snapshot are used to update the data structures *S[...] and R[...]* as follows. Let P_j be the destination for some message in the incremental send history. If this message appears in *R[i,j]*, then delete it from *R[i,j]*. Since this message has already been received, it is not in the channel according to the current cut. If the message does not appear in *R[i,j]*, then the message is appended to *S[i,j]*. An analogous procedure is followed for each message in *cut[i].increcv*.

4.2.2 Evaluating Channel Predicates

The second major activity of the checker is to evaluate unknown channel predicates. In Figure 4 the function *chanp(S[i,j])* is used for this purpose. A channel predicate is only evaluated for channels between two green states. Since those states are known to be concurrent, it is clear that the R list for the channel will be empty. All messages that have been received by P_j must have already been sent by P_i. Hence, the S list contains a sequence which exactly represents the state of the channel.

It should be noted that for many important channel predicates, the time to evaluate the channel predicate is constant. For example, the predicates, "The channel is empty", "The channel has k or more messages" and "The minimum time stamp of messages in the channel is at least k" can all be evaluated in constant time if appropriate data structures are used to represent the messages in the S list.

```
update-channels(i)

    /* for all messages sent by Pi to Pj */
    for (j : cut[i].incsend[j] ≠ ∅) do
        S′ := S[i,j];
        R′ := R[i,j];
        S[i,j] := S′ ⊕ (cut[i].incsend[j] - R′); /* concatenate */
        R[i,j] := R′ − cut[i].incsend[j];
        if (cp[i,j] = T ∨ cp[i,j] = F_r) cp[i,j] := X;
    endfor

    /* for messages received by Pi from Pj */
    for (j : cut[i].increcv[j] ≠ ∅) do
        S′ := S[j,i];
        R′ := R[j,i];
        R[j,i] := R′ ⊕ (cut[i].increcv[j] - S′); /* concatenate */
        S[j,i] := S′ − cut[i].increcv[j];
        if (cp[j,i] = T ∨ cp[j,i] = F_s) cp[j,i] := X;
    endfor
```

Figure 6: Procedure update-channels

5 Correctness of the Algorithm

Now that the algorithm for detection of a GCP has been given, the correctness of this algorithm will be shown. First, some properties of the program are given that will be used in demonstrating soundness and completeness.

Lemma 5.1 *The following is an invariant of the program assuming that the function paint-state is atomic.*

$$\forall i, j :: (cut[i].color = green) \wedge (cut[j].color = green) \Rightarrow cut[i] \| cut[j]$$

Proof: Initially, the invariant is true because $cut[i].color = red$ for all i. The color of $cut[i]$ is set to green only in the *paint-state* function. In that function, $cut[i]$ is compared with all $cut[j]$ whose *color* are green. If $cut[i]$ is not concurrent with $cut[j]$, then one of them is painted red preserving the invariant assertion after the *paint-state* function. □

The following lemma is crucial in making the detection algorithm efficient. It enables the algorithm to discard any red colored state.

Lemma 5.2 *For all i if $cut[i]$.color is red, then there does not exist any global cut satisfying the GCP that includes $cut[i]$.*

Proof: The proof is by induction on the number of states which have been painted red. The statement is true initially because $cut[i]$ is initialized to a fictitious state and there cannot exist a global cut that includes this state. Assume that the lemma is true for all states painted red so far. The variable $cut[i]$ is painted red either in function *paint-state* or after evaluation of a channel predicate. We consider both of these cases:

Case 1 $cut[i]$ is painted red in *paint-state* function. — This implies that there exists j such that $cut[i] < cut[j]$. We show that there is no state in process P_j which is a part of a global cut

with $cut[i]$ satisfying the GCP. From the program, a cut is advanced to the next state only if the current state is red. This implies that any predecessor of $cut[j]$ is red, and therefore, by our induction hypothesis, ineligible for a global cut satisfying the GCP. States $cut[j]$ and $cut[i]$ cannot be part of any global state since $cut[i] < cut[j]$. Further, by our assumption of FIFO between non-checker processes and the checker process, all states later than $cut[j]$ in the queue for P_j are greater than $cut[j]$ and so also greater than $cut[i]$. This implies that no other candidate states from P_j can be concurrent with $cut[i]$. Therefore, $cut[i]$ can be eliminated.

Case 2 $cut[i]$ is painted red during evaluation of channel predicates. — This implies that either $cp[j,i]$ is false and the predicate can not be made true by process j sending more messages ($cp[j,i] = F_s$), or that $cp[i,j]$ is false and can not be made true by process j receiving more messages ($cp[i,j] = F_r$). We show that there is no state in process P_j which is part of a global cut with $cut[i]$ satisfying the GCP. As in Case 1, any predecessor of $cut[j]$ is red, and therefore, ineligible for a global cut satisfying the GCP. States $cut[j]$ and $cut[i]$ cannot be part of any global state satisfying the GCP since the channel predicate is false along a cut including these states. This implies that the channel predicate is also false for $cut[i]$ and any successor of $cut[j]$. Therefore, $cut[i]$ can be eliminated.

□

The following lemma describes the role of $S[i,j]$ and $R[i,j]$. We use auxiliary variables $cut[i].Sent[j]$ and $cut[i].Rcvd[j]$. These variables are used only for the proof and not in the actual program. The variable $cut[i].Sent[j]$ is the sequence of all messages sent by the process i to process j until $cut[i]$. Similarly, $cut[i].Rcvd[j]$ is the set of all messages received by process P_i from process P_j until $cut[i]$. Note that these are complete histories, unlike *incsend* and *increcv* used in the program.

Lemma 5.3 *The following is an invariant of the program outside the body of while loop in Figure 4.*

$$\begin{aligned} S[i,j] &= cut[i].Sent[j] - cut[j].Rcvd[i] \\ R[i,j] &= cut[j].Rcvd[i] - cut[i].Sent[j] \end{aligned}$$

Proof: We prove the claim for $S[i,j]$. The proof for $R[i,j]$ is analogous. The invariant of Lemma 5.3 is initially true because all channels are empty and $S[i,j]$ is initialized to empty. Assume that it is true for all previous cuts. We show that on advancing the cut, the invariant stays true. Using the induction hypothesis we state:
$S'[i,j] = cut'[i].Sent[j] - cut'[j].Rcvd[i]$ and: $R'[i,j] = cut'[j].Rcvd[i] - cut'[i].Sent[j]$
by letting P_i be the process along which the cut is advanced. Further, let $cut'[i]$ be the previous state which has red color and $cut[i]$ be the new state. From the program text, $cut[i].incsend[j]$ contains all messages sent from P_i to P_j between $cut'[i]$ and $cut[i]$. Therefore,

$$cut[i].Sent[j] = cut'[i].Sent[j] \cup cut[i].incsend[j].$$

Also, $cut[j].Rcvd[i]$ has not changed. For proof purposes we use $\cup$ rather than $\oplus$, concatenation. Since the latter is stricter, the invariant holds for the algorithm. We now compute:

$$\begin{aligned} & cut[i].Sent[j] - cut[j].Rcvd[i] \\ &= (cut'[i].Sent[j] \cup cut[i].incsend[j]) - cut[j].Rcvd[i] \\ &= (cut'[i].Sent[j] - cut[j].Rcvd[i]) \cup (cut[i].incsend[j] - cut[j].Rcvd[i]) \end{aligned}$$

The last equality is derived by distributing the set subtraction over append. Therefore, from the induction hypothesis on $S'[i,j]$, it follows that

$$cut[i].Sent[j] - cut[j].Rcvd[i] = S'[i,j] \cup (cut[i].incsend[j] - cut[j].Rcvd[i]). \tag{1}$$

We now note that by the induction hypothesis on $R'[i,j]$,

$$R'[i,j] = cut[j].Rcvd[i] - cut'[i].Sent[j].$$

Therefore, by property of set subtraction:

$$R'[i,j] = cut[j].Rcvd[i] - (cut'[i].Sent[j] \cap cut[j].Rcvd[i]).$$

That is,

$$cut[j].Rcvd[i] = R'[i,j] \cup (cut'[i].Sent[j] \cap cut[j].Rcvd[i]).$$

We substitute this into Equation 1 to obtain:

$$\begin{aligned} & cut[i].Sent[j] - cut[j].Rcvd[i] \\ &\quad = S'[i,j] \cup (cut[i].incsend[j] - (R'[i,j] \cup (cut'[i].Sent[j] \cap cut[j].Rcvd[i]))) \\ &\quad = S'[i,j] \cup (cut[i].incsend[j] - R'[i,j]) \end{aligned}$$

The last equality follows from the fact that $cut[i].incsend[j]$ and $cut'[i].Sent[j]$ are disjoint. Note that *update-channels* performs an equivalent operation. □

The next Lemma shows that if $cp[i,j]$ has a value other than X, then it has the correct value of $chanp(cut[i].Sent[j] - cut[j].Rcvd[i])$.

Lemma 5.4 *For all i, j, if $cut[i]$ and $cut[j]$ are green, then*

$$(cp[i,j] \neq X) \Rightarrow cp[i,j] = chanp(cut[i].Sent[j] - cut[j].Rcvd[i])$$

Proof: Our proof is again based on induction on the cut. The assertion is true in the initial cut because $cp[i,j] = X$.

The assertion can turn false only when either the cut is advanced, or the value of $cp[i,j]$ is set to T, F_s, or F_r for the current cut. We do a case analysis.

Case 1 If the cut is advanced, a channel predicate can be affected only if some messages have been sent or received since the last evaluation. If the channel state has not changed due to sends, that is, $cut[i].incsend[j] = \emptyset$ then $cp[i,j]$ has not changed. Also, if $cut[j].increcv[i] = \emptyset$ then $cp[i,j]$ has not changed. Thus, the assertion is maintained. Now assume that a message has indeed been sent. If the previous value was T, then the new state is unknown and therefore $cp[i,j]$ is set to X. If the previous value was X, it is not changed and the assertion still holds. If the previous value was F_s, then the additional sends performed by process i can not make the predicate true. Hence it should stay F_s. Finally, if the previous value was F_r, $cp[i,j]$ is set to X. Thus, if a message is sent, the invariant is preserved. An analogous argument shows that the invariant is preserved when messages are received.

Case 2 The value of $cp[i,j]$ is set to a value other than X only if it is currently X and if both $cut[i]$ and $cut[j]$ are green. Furthermore the value is set by the expression: $cp[i,j] := chanp(S[i,j])$. From Lemma 5.3, this is equivalent to

$$cp[i,j] := chanp(cut[i].Sent[j] - cut[j].Rcvd[i]).$$

Thus, the invariant holds. □

We are now ready to prove that our algorithm is sound and complete. The next theorem says that our algorithm never makes a false detection. If the detect flag is true, then the current cut indeed satisfies the GCP.

Theorem 5.5 *(Soundness) If detect flag is true then there exists a cut in which the GCP is true. Moreover, the cut produced by the algorithm is the first cut for which the GCP is true.*
Proof: The *detect* condition evaluating to true is equivalent to $(\forall i :: cut[i].color = green) \wedge (\forall i, j :: cp[i, j] = T)$. By the algorithm of the non-checker process, $l_i(cut[i])$ holds. From Lemma 5.1 for all $i, j :: cut[i] || cut[j]$. It remains to be shown that all channel predicates are true. From the detect condition $\forall i, j :: cp[i, j] = T$. This implies that all channel predicates are true from Lemma 5.4. Thus, the cut satisfies the GCP.

We now show that this is the first such cut. Observe that the cut is advanced only when $cut[i]$ is red. From Lemma 5.2, $cut[i]$ cannot be part of any cut that makes the GCP true. Since all cuts previous to the current cut have at least one state red, it follows that the detected cut is the first cut in which the GCP is true. □

Theorem 5.6 *(Completeness) Let C be the first cut that satisfies the GCP. Then the GCP algorithm sets detect flag to be true with C as the cut.*
Proof: Since C is the first cut that satisfies the GCP, all the earlier states cannot make the GCP true. We show that all earlier states are painted red. The proof is by induction on the number of total states that are before this cut. If there are no states, then the claim is true. Assume that k states have been painted red. Consider the last state painted red. There is at least one more state ahead of it. This makes the while condition true and the cut is advanced to the next state. If this next cut is not equal to the cut C, then there exists at least one violation of the concurrency relation or channel predicate in the current cut. Therefore, for all cuts preceding C, at least, one state is painted red, and because of this, the cut will be advanced. Eventually, the checker will advance the cut to C. By Lemma 5.2, all states must be green. By Lemma 5.4, no CP-state flags can be set to F. Eventually, all CP-state flags will be set to T, since the checker can not enter the while loop. At this point, the checker will exit the repeat loop, and the detect flag will be set to true. □

5.1 Overhead analysis

We do overhead analysis only for the checker process. We use the following parameters:
N: Total number of processes in the system
n: processes involved in the GCP ($n \leq N$)
m: maximum number of messages sent/received by any process

Time complexity: There are three components to the computation of the checker process. The first two components are the functions *paint-state* and *update-channels* which are called when the cut is advanced. The third component is the evaluation of channel predicates. We describe the time complexity of each of these components.

Note that it takes only two comparisons to check whether two vectors are concurrent [14]. Hence, each invocation of *paint-state* requires $O(n)$ time steps. This function is called at most once for each state and there are at most mn states. Therefore, $O(mn^2)$ time is spent in the *paint-state* function.

The function *update-channels* performs two operations: subtracting and appending of message sequences. If the sequences are ordered, both of these operations require linear time in the size of the sequences. Since each process sends and receives at most m messages, the sum of the sizes of these sequences is $O(m)$. Therefore, the time spent in a single invocation of *update-channels* is $O(m)$. Since there are mn states, the total time spent in this function is $O(m^2n)$.

The value of a channel predicate can change only when a message is sent or received on the channel. Thus, there are at most two evaluations of the predicate per message. There are at most mn messages. If each predicate evaluation takes time proportional to the size of the channel, then each predicate evaluation is $O(m)$. Therefore, the total amount of time required to evaluate channel predicates is $O(m^2n)$.

Based on this evaluation, the time complexity of the checker process is $O(n^2m+m^2n)$. However, it should be observed that, in practice, the time complexity is much closer to $O(n^2m)$. First, the time required for *update-channels* is typically much smaller than $O(m^2n)$. In fact for FIFO channels, the total computation for *update-channels* is $O(mn)$. Second, evaluating a channel predicate is often a constant time operation as discussed in Section 4.2.2. In these cases, the total time spent by the checker process evaluating channel predicates is also $O(mn)$.

Space complexity: The main space requirement of the checker process is the buffer for the local snapshots. Each local snapshot consists of an *lcmvector* and incremental send and receive histories. The *lcmvector* requires $O(n)$ space. Note that strictly speaking, each *lcmvector* may require $O(n \log m)$ bits, but we assume that storage and manipulation of each component is a constant time/space overhead. This is true in practice because one word containing 32 bits would be sufficient to capture a computation with 2^{32} messages. Since there are at most $O(mn)$ local snapshots, $O(n^2m)$ total space is required to hold the component of local snapshots devoted to vector clocks.

Typically, evaluating a channel predicate does not require the entire contents of the messages. We assume that the relevant information from each message can be encoded in a constant number of bits. Hence the total space required for all incremental send and receive histories is $O(mn)$.

Therefore, the total amount of space required by the checker process is $O(mn^2)$.

Message Complexity: Every process sends $O(m)$ local snapshots to the checker process. Using the same assumptions as made for space complexity, it follows that $O(mn)$ bits are sent by each process.

6 Implementation and Experimentation

We have implemented our GCP detection algorithm using the Parallel Virtual Machine (PVM) message passing infrastructure [10]. PVM provides asynchronous sending/receiving of messages, mechanisms for spawning jobs on remote machines, and a distributed "signal" mechanism that allows Unix process signals to be delivered to remote processes. PVM is widely used for both parallel and distributed applications. Ports of PVM are available for most Massively Parallel Processors (MPPs) and networked workstations. We augment the standard PVM routines with vector clocks, and with the capacity to communicate with a central checker process. The interaction with the checker process is transparent to the programmer. The implementation is in the C++ programming language, and uses PVM version 3.1. We have tested our implementation on a homogeneous network of ten IBM RS/6000 model 350 workstations.

6.1 Distributed Termination Detection

We demonstrate the GCP detection algorithm with the distributed termination problem. The application consists of a set of interacting processes. Each process performs a random amount of work, and then, with probability p, sends a message to another (randomly selected) processor. The process then becomes idle and waits to receive an incoming message. The problem that must be solved is to determine when the computation has terminated. The condition for termination can be elegantly described as a GCP:

1. All processes are in the idle state — this condition is a conjunction of local predicates.
2. All channels are empty — this condition is a conjunction of a send monotonic channel predicate on each channel.

This problem has been well studied, and several solutions have been proposed. In this section we compare the GCP solution to two well known solutions, Dijkstra's algorithm [4] and Misra's algorithm [17]. Both of these approaches utilize a token that is continually passed along a predetermined cycle of the processes. In Dijkstra's algorithm, a simple ring is used. The token will visit each process every trip around the ring, but will not traverse every channel. Dijkstra's algorithm requires that message delivery be instantaneous.[1] In Misra's algorithm, the cycle includes every channel in the system. Hence, processes are visited by the token multiple times in a single cycle. Misra's algorithm is more expensive, but requires merely that individual channels are FIFO.

The application exhibits behavior that is linear in the number of messages sent by each process (the value of m). The execution time for the application is theoretically constant as the number of processes is increased. However since we are limited to only 10 physical processors, the execution time does increase as more processes are added to the system.

The overhead of Dijkstra's algorithm and Misra's algorithm is linear in the number of messages sent. With more messages sent, the token must take more cycles. However, since the processes are typically idle when they receive the token, the effective overhead increases only slightly as the number of messages is increased.

Since evaluating whether a channel is empty can be done in constant time, the GCP implementation has overhead that is linear in the number of messages for this application. This overhead is principally observed in the execution time of the checker process.

Dijkstra's algorithm is linear in the number of processes, since each process is visited at most once during each cycle of the token. However both Misra's algorithm and the GCP technique are quadratic in the number of processes.[2]

6.2 Experimental Results

We implemented all three algorithms and executed them on our 10-processor IBM RS/6000 cluster. The 10 machines share a dedicated ethernet subnetwork (10 Mbps). The data was collected while the machines were running in multi-user mode, but during periods when the machines were not in use for other purposes. The programs were compiled with the GNU C and C++ compilers (gcc version 2.5.8) using the -O optimization flag. PVM version 3.1 was used as the underlying message passing medium.

[1] The PVM 3.1 implementation uses UDP messages for interprocessor communication. However, when all processes are mapped to machines the same physical subnet, the aggregate behavior is that message delivery is instantaneous.

[2] Misra's algorithm is actually linear in the number of channels. However, on a fully connected system, there are $O(n^2)$ channels.

The test application is implemented as follows. The probability of a process sending a message is held fixed for the duration of the program. Messages are indistinguishable from each other. Processes do not discern from which process a message originated and do not examine the contents of messages. The behavior of a process is determined solely by the number of messages it receives and by the behavior of the random number generator. Our implementation is designed so that the distributed program is pseudo-random, but deterministic. Hence, by controlling the seed used for the random number generator we can faithfully reproduce the exact application behavior.

The amount of work performed between messages is set to zero in this implementation to emphasize the overhead generated by the distributed termination detection algorithm. In real code, the relative cost of this overhead would depend on the grain size of the computation, and would presumably be significantly lower.

Ten runs were performed with different random number seeds for each data point in figures 7 and 8. The data point is the average time from these ten runs. The same seeds were used for each of the three configurations; the GCP version, Misra's algorithm and Dijkstra's algorithm.

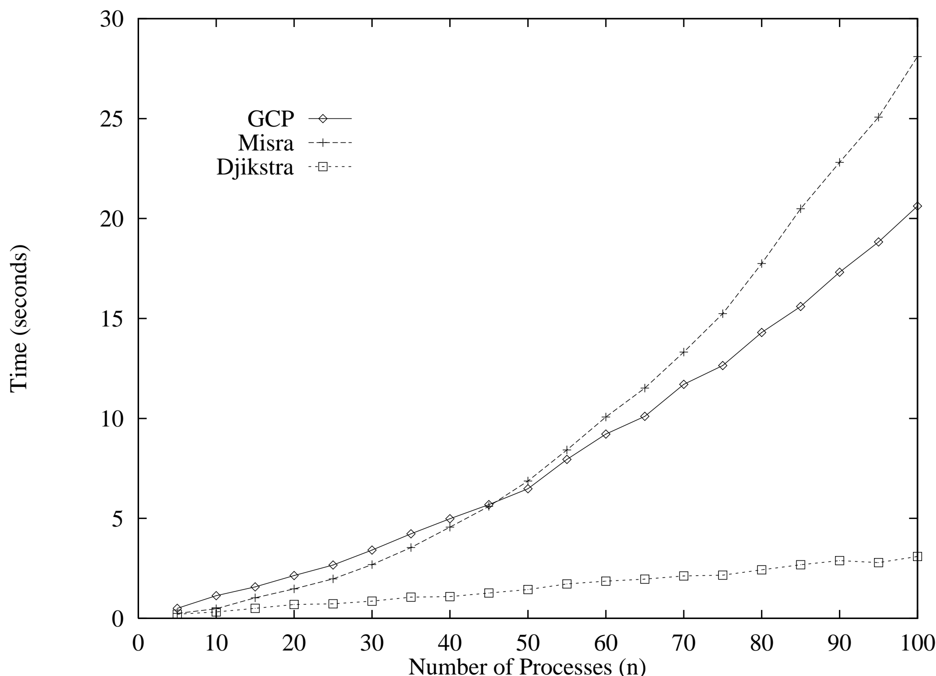

Figure 7: Time to Detect Distributed Termination as a Function of Processes

Figure 7 shows the performance of the three algorithms as the number of processes is increased. The data points with more than 10 processes were collected by spawning multiple processes on each processor. The probability of sending a message was set to 90% for each run. This value causes each process, on average, to send 9 messages before terminating. Since the computation is pseudo-random, the actual number of messages sent varies from run to run and from process to process. The GCP detection algorithm demonstrates comparable performance to Misra's algorithm over a large range of system sizes.

Figure 8 shows the performance of the three algorithms as the number of messages increases. We held the number of processes fixed at 10, one on each physical machine.[3] The X-axis in Figure 8

[3]For the GCP implementation, an additional process (the checker) is required. For this version, 11 processes are

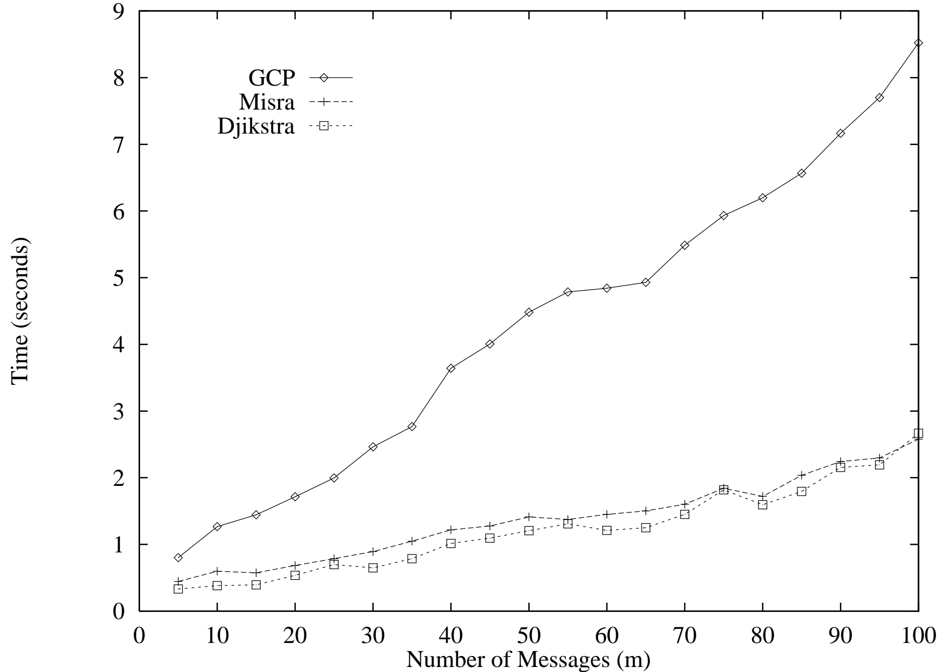

Figure 8: Time to Detect Distributed Termination as a Function of Messages

indicates the expected value of the number of messages sent by each process. Again, due to the pseudo-random nature of the application, the actual number of messages sent varies. The data points represent the average from 10 runs with different random number seeds. All three versions demonstrate time complexity that is linear in the number of messages sent. However, the cost per message for the GCP algorithm is significantly larger than that for either Dijkstra's or Misra's algorithm. In part, this larger cost is attributable to the quality of our implementation. A more highly optimized version should reduce this cost appreciably. However, even as it stands, we can argue that the performance of our implementation is reasonable for coarse grained computations. The slope of the line in Figure 8 is approximately 0.08 seconds per message.[4] Hence, if the amount of computation performed by an application significantly exceeds 80 milliseconds for each message sent, then the relative cost of the GCP detection algorithm will be negligible.

7 Conclusions

The ability to detect an arbitrary conjunction of local and channel predicates is sufficient to detect any global predicate that can be expressed using boolean connectives. We have presented a definition for Generalized Conjunctive Predicates and an algorithm for detecting an important class of these predicates, those with monotonic channel predicates.

The concept of monotonicity for channel predicates is useful for two important reasons. First, monotonicity is both a necessary and sufficient condition for the set of consistent cuts satisfying global properties to contain an infimum under the usual ordering. That is, the notion of the

used, and one machine hosts two processes.

[4]Recall, this figure is based on 10 processes. With more processes, the value would be higher.

first consistent cut satisfying a GCP is always well defined if and only if channel predicates are monotonic. Second, monotonicity allows an efficient algorithm to detect GCPs.

We have also presented an efficient algorithm to detect the first consistent cut in which a GCP is true. Our algorithm requires $O(n^2m + m^2n)$ time in the worst case and $O(n^2m)$ for many interesting problems. This algorithm has been implemented in a programming environment using PVM and demonstrated on a network of IBM RS/6000 workstations.

References

[1] Ö Babaoğlu, and K. Marzullo, "Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms," *Distributed Systems*, 2nd Edition, editor Sape Mullender, Addison Wesley, New York, NY. 1994, pp. 55-96.

[2] K. Chandy, and L. Lamport, "Distributed Snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, no. 1, pp. 63-75, February 1985.

[3] 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.

[4] E. W. Dijkstra, W. H. J. Feijen, and A. J. M. VanGasteren, "Derivation of a Termination Detection Algorithm for Distributed Computation," *Inf. Proc. Letters,*, Vol. 16, June 1983, pp. 217-219.

[5] C. J. Fidge, "Partial Orders for Parallel Debugging," *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging,* also *SIGPLAN Notices*, Vol. 24. No. 1. January, 1989. pp. 183-194.

[6] E. Fromentin, M. Raynal, V. K. Garg, and A. Tomlinson, "On the Fly Testing of Regular Patterns in Distributed Computations," *Proceedings of the 23rd Int. Conference on Parallel Processing*, Pennsylvania State University, August 1994.

[7] V. K. Garg, and A. Tomlinson, "Using Induction to Prove Properties of Distributed Programs," *Proceedings of the Symposium on Parallel and Distributed Processing*, Dallas, Texas, December, 1993, pp.478-485.

[8] V. K. Garg, and B. Waldecker, "Detection of Weak Unstable Predicates in Distributed Programs," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 3, March 1994, pp. 299-307.

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

[10] A. Geist, et. al., *PVM 3 Users' Guide and Reference Manual*, Oak Ridge National Laboratory, May, 1993.

[11] D. Haban and W. Weigel, "Global events and global breakpoints in distributed systems," *Proc. of the 21st 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," *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging,* San Diego, California, May, 1993.

[13] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, pp. 558-565, July 1978.

[14] 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.

[15] Y. Manabe, and M. Imase, "Global Conditions in Debugging Distributed Programs," *Journal of Parallel and Distributed Computing*, Vol. 15, pp. 62-69, 1992.

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

[17] J. Misra, "Detecting Termination of Distributed Computation Using Markers," *Proc. of the 2nd annual ACM Symposium on Principles of DC*, Aug, 1983, pp. 290-294.

[18] 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.

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

[20] 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, pp. 21–31.

[21] *MPI: A Message-Passing Interface Standard*, Message Passing Interface Forum, May, 1994.