Efficient Detection of Channel Predicates in Distributed Systems[1]

V. K. Garg C. M. Chase Richard Kilgore J. Roger Mitchell

January 19, 1995

[1]This paper is an extended and substially revised version of a paper which appeared as V. K. Garg, C. M. Chase, J. R. Mitchell, and R. Kilgore, “Detecting conjunctive Channel Predicates in a Distributed Programming Environment,” *Proc. of the Twenty-Eighth Hawaii International Conference on System Sciences*, January 1995, Vol. II, pp. 232 – 241, 1995.

Abstract

This paper discusses efficient detection of global predicates in a distributed program. Previous work in this area required predicates to be specified as a conjunction of predicates defined on individual processes. Many properties in distributed systems, however, use the state of channels such as “the channel is empty,” or “there is a token in the channel”. In this paper, we introduce the concept of a *monotonic* channel predicate and provide efficient centralized and distributed algorithms to detect any conjunction of local and monotonic channel 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 algorithms 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.

Index Terms:

1. Distributed Systems
2. Distributed Debugging
3. Predicate Detection
4. Channel Predicate
5. Monotonic Channel Predicates

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 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 [CL85, CM91, FRGT94, GW94, GW92, HW88, Lam78, Mat89, MI92, MC88, TG93]. See [BM94, SM94] 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 [CL85] for detecting a global predicate involves periodically taking a global snapshot of the state of the system. If the predicate becomes true at some time t, their algorithm will eventually create a snapshot using states after t. Since the predicate they are detecting is stable, the predicate must still be true, and will be detected by their algorithm.

Unlike stable predicates, unstable predicates may alternate between true and false values. Cooper and Marzullo [CM91] presented a method for detecting general predicates which included unstable predicates. Their approach, though, requires exponential ($O(m^n)$) time where m is the maximum number of events a monitored process has executed and n is the number of processes. In this paper, we define a subclass of predicates which can be detected in low-order polynomial time.

Manabe and Imase [MI92] 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 [GW94] 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 [GW94] 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 classic 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*. An example of a monotonic predicate is, “The channel contains exactly 5 messages”. When the channel contains less than 5 messages, this predicate is false and it will remain false until more messages are sent on the channel. If there are more than 5 messages in the channel, then the predicate is false, and it will remain false until some messages are received on the channel. We show that 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 a 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 conjunction of channel predicates can only be well defined when monotonic channel predicates are used. A formal definition of monotonic channel predicates is given in Section 2.

The next section will present the notation, definitions of predicates, and our model of a distributed system, which are necessary in understanding the method of detecting conjunctive channel predicates. Section 3 presents two predicate detection algorithms. The first algorithm, described in Section 3.2 is based on a centralized predicate checker. The second algorithm (Section 3.3) is fully distributed, with the required data structures evenly divided among the N processes. Section 4 summarizes the paper.

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 message-passing distributed 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*. Program actions, such as changing the value of a variable, sending or receiving a message, occur during the transitions between these states. That is, the process P_i generates the trace $r[i] = \alpha_{i,0}\alpha_{i,1}\ldots\alpha_{i,m}$, where α_i’s are the local states, and where m is the maximum number of distinct states in a single process. There are three classes of program actions (hereafter *events*) that can occur between these states — sending of a message, reception of a message or some internal event. Finally, the state of a process is defined by the value of all its variables including its program counter.

The distributed system also includes a finite set of uni-directional channels. In this paper we will assume there are N^2 channels arranged as a fully-connected network. We label the channels as an $N \times N$ matrix with channel(i, j) used for messages sent by process P_i to P_j. However, our algorithms can be trivially extended to work with any number or organization of channels as long as the network is connected.

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

2.2 Global Predicates

Typically, one is interested in determining if some predicate becomes true during the execution of a program. Since the predicate can potentially be some function of the state of every process and channel in the system, we must identify some reasonable definition of simultaneity between process states. For this, we use the happened-before relation of Lamport[Lam78]. The happened-before relation for two process states α and β can be formally stated as: $\alpha \rightarrow \beta$ iff:

1. $\alpha \prec \beta$ where $\prec$ means occurred before in the same process, or
2. $\alpha \leadsto \beta$ where $\leadsto$ 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. $\exists \gamma : \alpha \rightarrow \gamma \wedge \gamma \rightarrow \beta$.

Two states for which the happened-before relation does not hold in either direction are said to be concurrent. The symbol, $\|$, is used to represent concurrency. The relationship can be formally stated as:

$$\alpha \parallel \beta \Leftrightarrow (\alpha \not\rightarrow \beta \;\wedge\; \beta \not\rightarrow \alpha)$$

Given a set of N states, one from each process, if this condition holds for all pairwise combinations of the states, then this set forms a *consistent cut*. Thus, the global predicate detection problem then becomes one of finding a consistent cut in which the predicate evaluates to true.

2.2.1 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 detect a local predicate on its own.

2.2.2 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 the send events and receive events are performed by different processes, no single process can evaluate a channel predicate on its own.

We use the following notation to indicate the accumulation of send and receive events on a channel.

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 P_i to P_j.
$\beta.Rcvd[i]$:	sequence of all messages received at or before state β from P_i to P_j.

A channel predicate can then be written as:

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

or in short notation as:

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

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.

We require channel predicates to be at least monotonic. The requirement for monotonicity can be stated formally as:

Definition 2.1 *A channel predicate, chanp(S), is said to be* monotonic *iff, for any sets of messages, S, S', R :*

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

That is, given any set of send events, *S*, that causes the predicate to be 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.

The following Lemma follows directly from the definition of monotonicity:

Lemma 2.2 *For any states* $\alpha, \beta, and\ \gamma$

$$\beta \prec \gamma \wedge chanp(\alpha, \beta) = F_r \Rightarrow chanp(\alpha, \gamma) = F_r$$

Proof: Since the state of the sender has not changed, the predicate will remain F_r. □ *Similarly:*

$$\alpha \prec \gamma \wedge chanp(\alpha, \beta) = F_s \Rightarrow chanp(\gamma, \beta) = F_s$$

Example 1. *Empty Channel:* $chanp(S) \equiv (S = \emptyset)$: In any state in which this predicate is false, sending more messages will not make it true. This predicate can be used in termination detection.

Example 2. *Channel Overflow:* $chanp(S) \equiv (\sum_{m \in S} \text{sizeof}(m) \geq k)$: In any state, if the channel is not currently full, then receiving more messages cannot make it full.

Example 3. *Exactly k Messages in Channel:* $chanp(S) \equiv (|S| = k)$: In any state where there are more than k messages in the channel, this predicate cannot be made true by sending more messages. In any state when there are less than k messages in a channel, this predicate cannot be made true by receiving more messages.

2.2.3 Generalized Conjunctive Predicate

We call a predicate detected by our algorithm a generalized conjunctive predicate (GCP). A GCP is formed by any collection of local predicates and channel predicates. The GCP is true if and only if all of its component predicates are simultaneously true in a consistent cut.

For example, termination detection can be easily represented as a GCP detection problem. For each of the N processes, a local predicate is defined as “The process is idle”. For each of the $(O(N^2))$ channels, a channel predicate is defined as “The channel is empty”. Termination is equivalent to this GCP being satisfied. Another example is satisfying a lower bound, K, on the global virtual time of a distributed simulation. For each of the N processes, a local predicate is defined as “The local time is at least K”. For each channel, a channel predicate is defined as “The minimum time stamp over all messages in the channel is at least K”. The simulation has passed global virtual time K is equivalent to the GCP being satisfied. It should be noted that although both of these examples are stable predicates, our algorithms can also detect unstable GCPs.

The following 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 2.3 *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 $\text{chanp}_{ij}(E[i], E[j])$ 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 since C and D are consistent cuts, $\forall i, j \in I : E[i] || E[j]$ and $\forall i, j \in J : E[i] || E[j]$. We now show that $\forall i \in I, j \in J : E[i] || E[j]$. Assume $E[i] \rightarrow E[j]$. Substituting, we have $C[i] \rightarrow D[j]$.

However, since $j \in J$, we know $D[j] \preceq C[j]$, leading to $C[i] \rightarrow C[j]$, a contradiction. Therefore $E[i] \not\rightarrow E[j]$. A symmetric argument shows that $E[j] \not\rightarrow E[i]$. Hence, E is a consistent cut.

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

   Case 1 $chanp_{ij}(E[i], E[j]) = F_s$ — Since $E[i] \preceq D[i]$, from Lemma 2.2 we know that $chanp_{ij}(D[i], E[j]) = F_s$. Hence $chanp_{ij}(D[i], D[j])$ is false, a contradiction.

   Case 2 $chanp_{ij}(E[i], E[j]) = F_r$. — Similarly, since $E[j] \preceq C[j]$, it follows that $chanp_{ij}(C[i], C[j])$ 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 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 uniquely 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 2.4 *The first cut that satisfies the GCP is always well defined only if all channel predicates in the GCP are restricted to 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 satisfies the GCP.

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

3 GCP Detection

The method of detecting the GCP is divided among monitor and application processes. The application processes are those processes which were used in the original computation (i.e., the program we are trying to debug). The GCP is defined over the state of the application processes and the state of the channels between application processes. The monitor processes are additional processes which are created solely for the purpose of predicate detection.

We present two efficient algorithms for monotonic GCP detection. The first algorithm uses a centralized monitor process to find consistent cuts and to evaluate all channel predicates. The second algorithm uses N monitor processes. Each monitor process evaluates at most N channel predicates and they collectively determine when a cut is consistent. Both algorithms distribute the task of evaluating local predicates.

3.1 GCP Algorithm: The Application Processes

We assign to each application process three functions related to predicate detection:

1. Identification of which states on other application processes happen before states on this application process.

2. Identification of states on this application process in which all local predicates are true.

3. Collection and delivery to the monitor process(es) of sufficient information to determine the state of any channel incident to this process.

To satisfy the first requirement, our algorithm uses vector clocks [Mat89, Fid89]. Each application process maintains a vector clock as part of the state of the process. The vector clock is attached to all messages sent between application process and 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 their respective vector clocks at these states

Each application process is assumed to be able to satisfy the second requirement trivially. Any state in which local predicates are not true is ignored. We can disregard many of the states in which local predicates are true as well. Only the first state following each send or receive event in which the predicate is true can be part of the *first* cut to satisfy a GCP. Since our predicate detectors are capable of finding exactly this cut, we can safely disregard all other states (i.e., we won't fail to detect the GCP if it occurs). Thus, there are at most M states of interest from each application process (recall that M is the maximum number of message events by any one process). For each of these states, the application process must construct a "local snapshot". This local snapshot is placed into a message and sent to a monitor process. The local snapshot includes the current vector clock from the application process. This information will allow the monitor process(es) to determine which local snapshots from other application processes are concurrent with this one.

To satisfy the third requirement, the application process also includes in the local snapshot an incremental record of activity on all channels incident to this process. For example, if the process has sent 2 messages and received one message since the last local snapshot was created, then the next local snapshot will contain a record of each of these events. Conceptually, a copy of the entire message is placed into the snapshot for all send events. However, in practice much less information is actually required. For example, if the predicate is "the channel is empty", then one bit will suffice, since the channel is empty precisely when the number of send events on the channel is equal to the number of receive events. This issue is addressed in more detail when the monitor processes are described. Note that for receive events, all that is necessary is that the sequence number of the message be placed into the snapshot.

We label the N application processes $P_1, \ldots, P_N$. Figure 2 shows the extensions we require to the behavior of each application process. We believe that the probe effect caused by this additional work is tolerable for most applications. Reducing or eliminating probe effect is an area of active research that is beyond the scope of this paper. It should be noted that the same extensions are required for both the centralized detector and the distributed detector. The only difference that is required is that for the centralized algorithm, all application processes send their local snapshots to the same monitor process (denoted M_0), whereas in the distributed algorithm each application process P_i sends its local snapshots to monitor process M_i.

3.2 Centralized GCP Algorithm

In the centralized algorithm, a single monitor process is responsible for searching for a consistent cut that satisfies the GCP. We label this process as M_0. 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, M_0 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 monitor process can then advance the cut by considering the successor to one of the eliminated states on the cut. If M_0 finds a cut for which no state can be eliminated, then that cut satisfies the GCP and the detection algorithm halts.

Figure 4 shows the algorithm used by M_0 to detect the GCP. The algorithm consists of a number of actions, each of which is guarded by some clause. Each action is assumed to be atomic. If more than one guard is true simultaneously, then the action that is performed can be selected non-deterministically. Some

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

initially $\forall j : j \neq i$ :vclock[$j$] = 0;
    vclock[$i$] = 1;
    firstflag = true;
    incsend = increcv = $\emptyset$;

for sending m do
    send (vclock, m);
    vclock[$i$]++ ;
    firstflag := true;
    incsend := incsend $\cup$ $\{$m$\}$;

upon receive (msg_vclock, m) do
    foreach j do:
        vclock[j] := max(vclock[j], msg_vclock[j]);
    done
    firstflag := true;
    increcv := increcv $\cup$ $\{$m$\}$;

upon (local_pred = true $\wedge$ firstflag) do
    firstflag := false;
    send (vclock, incsend, increcv) to monitor process;
    incsend:=increcv:=$\emptyset$;
```

Figure 2: Extensions to Application Process P_i for GCP Detection

constant-time performance gains can be realized by prioritizing the actions appropriately. Such optimization is beyond the scope of this paper. The algorithm terminates when none of the guards are true. When this occurs, the GCP has been detected, and the array state[1..n].vclock indicate which application process states are part of the cut. As an obvious extension, if some application process has terminated and all of the states from that process have been eliminated, M_0 can abort the detection algorithm.

3.2.1 Data Structures

The monitor process receives local snapshots from application processes. These messages are used by M_0 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 messages, those data structures that describe the state of the application processes, and those data structures that include information describing the state of the channels.

Incoming Message Queues The monitor process relies on being able to selectively receive a message from a specific application process. For example, at some phase in the algorithm M_0 may ask to receive a message sent specifically by P_i. Furthermore, we require that messages sent by an individual application process to the monitor process be received in FIFO order. If the message passing system does not provide

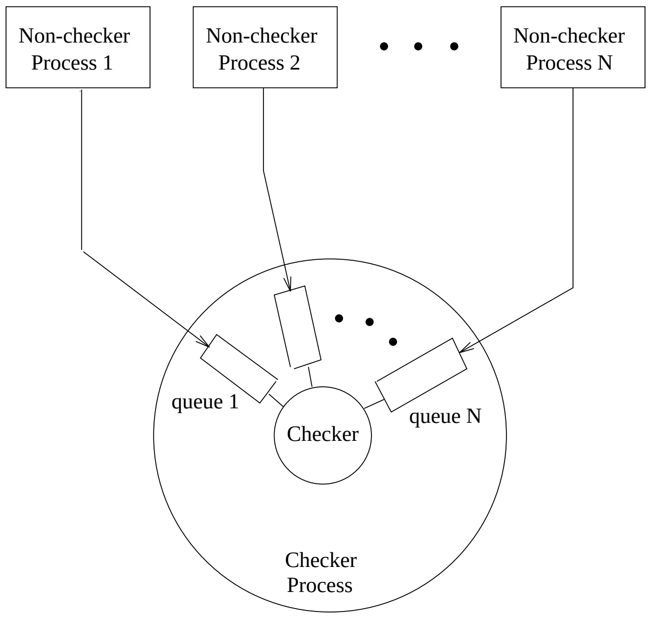

Figure 3: The Centralized GCP detector.

this support, it can be easily constructed using a set of queues.

We model the message passing system as a set of n FIFO queues, one for each application process over which some term of the GCP is defined. Recall that N was the total number of application processes in the system. Hence, $n \leq N$. We use the notation *q[1..n]* to label these queues in our algorithm.

Per-Process Data The monitor process maintains information describing one state from each application process P_i. The collection of this information is organized into a vector:

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

The *process_data* structure consists of a local snapshot (see Section 3.1) 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 red only if it cannot be part of a consistent cut that satisfies the GCP.

Per-Channel Data The monitor process maintains three data structures for each channel:

- *S[1..n, 1..n] : set of messages* — The pending-send set (or "S" set). The set 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] : set of messages* — The pending-receive set (or "R" set). The set contains each message that has been received from the channel, but not yet sent according to the current cut. Since the current

```
S[1..n,1..n], R[1..n,1..n] : sequence of message;
CP[1..n,1..n] : $\{F_s, F_r, T\}$;
state : array[1..n] of struct {
     v : vector of integer;
     color : {red, green};
     incsend, increcv : sequence of messages }
initially
     state[i].v = $\underline{0}$; state[i].color = red; S[i,j] = $\emptyset$; R[i,j] = $\emptyset$; CP[i,j] = $chanp_{ij}(\emptyset)$

/* advance the cut */
A1: upon ($\exists$ i : state[i].color = red) do
     state[i] := receive(q[i]);
     state[i].color := green;
     update_channels(i);

/* eliminate states which happened before other states */
A2: upon ($\exists i, j$ : state[i].color = green $\wedge$ state[i].vclock $<$ state[j].vclock)
     state[i].color := red

/* force more messages to be sent when channel is $F_r$ */
A3: upon (CP[i,j] = $F_r$ $\wedge$ state[i].color = green)
     state[i].color := red;

/* force more messages to be received when channel is $F_s$ */
A4: upon (CP[i,j] = $F_s$ $\wedge$ state[j].color = green)
     state[j].color := red;
```

Figure 4: Centralized GCP Detection Algorithm, Monitor Process M_0

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 set is empty.

- *CP[1..n, 1..n] : $\{F_s, F_r, T\}$* — 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, M_0 can avoid unnecessarily recomputing channel predicates by recording which predicates have remained true or false since the last time the predicate was evaluated.

3.2.2 Advancing the Cut

In any cut in which the GCP is false, we know that there must exist at least one state along the cut that can be eliminated. A formal representation of elimination is that:

Definition 3.1 *Given any cut C for which the GCP is false, a state $\alpha \in C$ can be labeled red, iff $\forall D$ for which the GCP is true, $C \leq D \Rightarrow \alpha \notin D$*

The algorithm works by considering states from each application process in sequence. Once a state has been labeled red, we must receive a new state from that process. We update the state of the S and R sets based on any message activity that occurred since the last snapshot. The procedure, *update_channels*, is used to update the channel state information. This procedure is shown in Figure 5.

A local snapshot contains a list of send events and a list of receive events. For each send event, update_channels first checks to see if the receiver is known to have already received this message. If so, the message is removed from the R set for the channel. If not, then the message is added to the S set for the channel. This latter case corresponds to the message still being in route.

```
update_channels(i)

    foreach message m sent by $P_i$ to $P_j$ do
        if (m $\in$ R[i,j]) R[i,j] := R[i,j] $-$ {m}
        else S[i,j] := S[i,j] $\cup$ {m}
        CP[i,j] := chanp$_{ij}$(S[i,j]);
    done

    foreach message m received by $P_i$ from $P_j$ do
        if (m $\in$ S[j,i]) S[j,i] := S[j,i] $-$ {m}
        else R[j,i] := R[j,i] $\cup$ {m}
        CP[j,i] = chanp$_{ij}$(S[i,j]);
    done
```

Figure 5: Procedure update_channels

3.2.3 Eliminating States Based Upon Causality

The GCP is true only if the cut is consistent. Since our algorithm is based on eliminating all predecessors to the first cut that satisfies the GCP, we should eliminate the older of any two states which are causally related. Action A2 performs this task.

3.2.4 Eliminating States Based Upon Monotonicity

Whenever a monotonic channel predicate is false, we know that either more messages must be sent, or more messages must be received in order for the predicate to become true. Actions A3 and A4 are based upon this fact. If the channel predicate is F_r, then the state from the sender can be eliminated since at least one more message must be sent before the predicate can become true. Action A3 labels the state from the sending process red. Action A4 performs an analogous activity for any channel whose predicate evaluates to F_s.

3.2.5 Evaluating Channel Predicates

Channel predicates can safely be evaluated at any time without affecting either the correctness or the worst-case time complexity of our algorithm. The S set always contains a list of messages that would be in the channel if every application process had executed exactly up to the current cut. Note that the R set may not be empty if this cut is not consistent. If the R set does contain some message m, then m is not in the channel (m has already been received), nor will it be in the channel at any time in the future.

3.2.6 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 correctness. The following lemma describes the role of $S[i,j]$ and $R[i,j]$. We use auxiliary variables $state[i].Sent[j]$ and $state[i].Rcvd[j]$. These variables are used only for the proof and not in the actual program. The variable $state[i].Sent[j]$ is the set of all messages sent by P_i to P_j prior to P_i reaching the state $state[i]$. Similarly, $state[i].Rcvd[j]$ is the set of all messages received by P_i from P_j prior to P_i reaching the state $state[i]$.

Lemma 3.2 *The following is an invariant of the program:*

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

Proof: The proof is by induction on the number of local snapshots received. The lemma is obviously true initially, since both *S[i,j]* and *R[i,j]* are initialized to $\emptyset$. Assume that the lemma holds for all snapshots received so far. We show that update_channels causes the lemma to hold for *S[i,j]* when one more snapshot is received from process P_i. The proofs for snapshots received from process P_j and for *R[i,j]* are analogous. Let *state[i]* denote the state from P_i that immediately precedes the snapshot, and *state′[i]* denote the state after the snapshot. Similarly, let *S[i,j]* be the value before the snapshot was received and let $S'[i,j]$ denote the value after the snapshot is received. We therefore wish to show:

$$S'[i,j] = state'[i].Sent[j] - state[j].Rcvd[i]$$

Since snapshots arrive in FIFO order, the following two identities hold:

$$\begin{aligned} state'[i].Sent &= state[i].Sent \cup incsend \\ state[i].Sent \cap incsend &= \emptyset \end{aligned}$$

We can see from the program that:

$$S'[i,j] = S[i,j] \cup (incsend - R[i,j])$$

By the induction hypothesis:

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

Hence, by substitution we have:

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

□

The following is also an invariant of the algorithm maintained by update_channels. The proof follows from Lemma 3.2.

Lemma 3.3 *CP[i,j] = chanp$_{ij}$(state[i], state[j])*

Theorem 3.4 *Let H be the first cut that satisfies the GCP. Then the centralized GCP algorithm terminates with $state[i] = H[i]$.*

Proof: We complete this proof in two parts. First we show that if state[1..n] is a predecessor to *H*, then at least one *state[i]* will be set to red. Since *H* is the first cut to satisfy the GCP, we know that either *state[1..n]* is not consistent or a channel predicate must be false. If *state[1..n]* were not consistent, then by the property of vector clocks, the guard for Action A2 must be true. Hence at least one *state[i]* will be set to red, a contradiction. If on the other hand, a channel predicate were false, then by Lemma 3.3 CP[i,j] must be either F_s

or F_r. Thus either action A3 or A4 would occur, and a state would be painted red. Therefore, if *state[1..n]* is a predecessor to *H*, the algorithm makes progress.

We now show that if $state[i] \in H$, then *state[i]* will not be labeled red. This condition guarantees we will not bypass the cut *H*. The proof is by induction on the number of states painted red. Assume that no element of *H* has been painted red so far. Cuts can be labeled red by actions A2, A3 and A4. We consider each case and show by contradiction that *state[i]* cannot be labeled red if $state[i] \in H$.

Case 1: Action A2 labels *state[i]* red. This implies that *state[i]* happened before some other state *state[j]*. By the induction hypothesis, $state[j] \preceq H[j]$. This leads to $H[i] \rightarrow H[j]$, a contradiction since H is a consistent cut.

Case 2: Action A3 labels *state[i]* red. This implies that for some j, CP[i,j] = F_r. By the induction hypothesis, *state[j]* must be either on *H*, or a predecessor to *H*. By Lemma 2.2 the predicate will therefore also have the value F_r at *H*, a contradiction since the GCP is satisfied by *H*.

Case 3: Action A4 labels *state[i]* red. This implies that CP[j,i] = F_s. Using similar reasoning as for Case 2, this implies that the channel predicate will be F_s along the cut *H*, a contradiction.

Hence, no component of *H* is ever painted red, and all predecessors to *H* are eventually painted red. Thus, our algorithm will eventually advance to the cut *H*. At this time, all guards are false and the algorithm will halt. □

3.2.7 Overhead Analysis

We do overhead analysis only for M_0. 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 application process
- s: the size of the largest message sent by any application process.

We also make the following simplifying assumption: a channel predicate can be evaluated in time proportional to the number of messages in the channel. This assumption holds for most predicates of interest.

Time complexity: Note that Action A1 can be performed at most mn times, since there are at most mn states. Each of the actions A2, A3 and A4 may also be applied at most mn times, since each of these actions labels a state red. Each state is made green initially by A1, and can only be labeled red once. We consider the complexity of each action in turn.

The work to perform Action A1 is determined by the cost to receive local snapshots plus the cost to update the channel states. Each local snapshot consists of a vector clock with n elements plus the incremental send and receive histories. Hence, the total number of bits from all local snapshots is bounded by $O(mn(n + s))$. The work performed in *update_channels* is dominated by the time to evaluate channel predicates. Each channel predicate must be evaluated at least once (for empty channels at the initialization of the system), and up to mn re-evaluations may be required. At any given time, there can be at most $O(m)$ messages in any channel (although, in practice there are typically much fewer). Thus $O(n^2 + m^2n)$ work is required to evaluate channel predicates. Therefore, Action A1 requires $O(n^2m + m^2n + mns)$ work.

The work required to perform the actions A2, A3 and A4 is constant time. However, the guards for these actions must also be evaluated. It must be noted that an implementation of our algorithm would not follow Figure 4 literally. Consider the guard for A2. Although at first glance it may appear that quadratic time is necessary for each evaluation of the guard, it can actually be tested in linear time. Assume that it is known that A2 does not apply. There is no need to test A2 again until Action A1 has occurred and at least a new state has been received. If *state[i]* is that new state, then A2 could apply only if $state[i] \rightarrow state[j]$ or $state[j] \rightarrow state[i]$ for some other state[j]. Hence it is only necessary make n comparisons of the vector clock[1] to know if A2 now applies. Finally, since A1 can occur at most mn times, the total amount of work for Action A2 is $O(n^2m)$.

Using two linked lists, Actions A3 and A4 can be tested in constant time. All channels whose predicates are F_r and whose sending process is currently green are kept in one such list, and all channels whose predicates are F_s and whose receiving process is green are in kept in the other. Obviously one of A3 or A4 applies iff its corresponding list is non-empty. The lists can be superimposed on the *CP[i,j]* array. Thus, inserting or removing channels from the list can be performed in constant time.

We conclude that the time complexity of the centralized algorithm is:

$$O(n^2m + m^2n + mns)$$

It should be noted this bound is fairly conservative. For example consider buffer overflow or termination detection as examples. In either of these cases, the evaluation of a channel predicate requires simply knowing

[1] Vector clocks can be compared in constant time.

how many bits remain in the channel. Hence, local snapshots do not need to include a copy of the message in the message histories, the S and R sets can be replaced by simple counters, and channel predicates can be evaluated in constant time. Thus, for these predicates, the time complexity is

$$O(n^2 m)$$

.

Space complexity: The main space requirement of M_0 is the buffer for the local snapshots. This space is $O(mn(n+s))$. Note that strictly speaking, each vector clock may require $O(n \log m)$ bits. This would increase the space complexity to $O(mn(n \log m + s))$. However, 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.

Message Complexity: Every of the n processes send at most m local snapshots to M_0. Each local snapshot contains $O(n+s)$ bits, for a total of total of $O(n^2 m + mns)$ bits communicated by the algorithm.

3.3 Distributed GCP Algorithm

This section describes a distributed version of the GCP detection algorithm. We use N monitor processes, denoted $M_1, \ldots M_N$. Each monitor process is paired with one of the N application processes. Whereas in the centralized algorithm, all application processes send their local snapshots to a single monitor process (M_0), in the distributed algorithm, each application process P_i sends its snapshots to monitor process M_i. It should be noted, that in a distributed debugger, no messages may actually be required for messages between P_i and M_i. The most reasonable implementation is to locate P_i and M_i on the same physical processor. In this case, M_i may be able to access local snapshots directly. (e.g. with the Unix *Ptrace* facility).

In the description of the algorithm we will refer to “monitor messages”. A monitor message is a message sent between monitor processes. A local snapshot (sent between an application process and a monitor process) is not a monitor message.

Figures 6 and 7 show the algorithm used by monitor process M_i.

3.3.1 Data Structures

We use the notation $M_i.x$ to indicate the value of local datum x on monitor process M_i. Most of the data structures in the distributed algorithm are directly related to data structures in centralized algorithm (see Section 3.2.1). The most recently received snapshot from P_i (previously *state[i]*) is stored in $M_i.state$. Each monitor process M_i is responsible for those channels on which P_i can send messages. The outstanding send list for channel$_{ij}$ (previously *S[i,j]*)is stored in $M_i.S[j]$. Similarly the outstanding receive list (previously *R[i,j]*) for that channel is $M_i.R[j]$, and the value of the channel predicate is recorded in $M_i.CP[j]$.

Since M_i does not have access to the receive events that occur on channel$_{ij}$, acknowledgment messages are required. We call the acknowledgment messages *delayed acknowledgment* (or *dack)* messages to emphasize the fact that the acknowledgment for some message is not sent immediately after the message is received. Consider some application process P_j that receives a message immediately before entering some state α. Let P_i be the application process that sent the message. Then monitor process M_j will eventually send a *dack* message to M_i. However, the *dack* is not sent until all predecessors to α have been eliminated by M_j.

Four data structures are related to the *dack* messages, and their use in maintaining the S and R sets. Each of these data structures is implemented as an array, with one entry per channel. The data structures are:

- M_i.dacks_sent[j] — a count of the number of *dacks* sent from M_i to M_j for channel$_{ji}$.
- M_i.dacks_rcvd[j] — a count of the number of *dacks* received by M_i from M_j for channel$_{ij}$.
- M_i.dacks_required[j] — a count of the minimum number of messages which must be received by P_i on channel$_{ji}$ before the GCP can be true.
- M_i.dack_pending[j] — a boolean flag which if true means that M_i is certain to receive at least one more *dack* message from M_j for channel$_{ij}$

Dack messages are one of two types of monitor messages. The other type of monitor message is a *dack_request* message. *Dack_request* messages are sent when a channel predicate is F_s, and it is known that more messages must be received in order for the channel predicate to become true. The use of these messages is described in detail below.

3.3.2 Termination

The distributed algorithm terminates when all M_i have terminated (i.e. all guards in Figure 6 are false), and all monitor messages have been received. We use a variation of Dijkstra's and Scholten's termination detection algorithm for diffusing computations [DS80]. GCP detection is not a true diffusing computation, since there is no single parent to the monitor processes. However, it is trivial to extend Dijkstra's algorithm to our needs by arbitrarily declaring M_1 as the parent of all other monitor processes and initializing the termination detection data structures accordingly. Thus, M_1 will detect termination. It should be noted that Dijkstra's algorithm is optimal in the number of messages sent for termination detection (equal to the number of monitor messages, which we will show is at most $2mn$).

When termination has been detected, the cut defined by the M_i.state variables is the first consistent cut for which the GCP is true.

```
A1: **upon** *my_color* = red
        *state* := receive snapshot from $P_i$
        *my_color* := green;
        update_channels(*state*);

A2: **upon** $\exists j$ : R[$j$] $\neq \emptyset$ $\wedge$ *my_color* = green
        *my_color* := red;

A3: **upon** $\exists j$ : CP[$j$] = $F_r$ $\wedge$ *my_color* = green
        *my_color* := red;

A4: **upon** $\exists j$ : CP[$j$] = $F_s$ $\wedge \neg$*dack_pending*[$j$]
        *dack_pending*[$j$] := true;
        send *dack_request*(*dacks_rcvd*[$j$]+1) to $M_j$;

A5: **upon** receive *dack_request*(count) from $M_j$
        *dacks_required*[$j$] := max(*dacks_required*[$j$],*dack_request*.count);

A6: **upon** $\exists j$ : *dacks_required*[$j$] $>$ *dacks_sent*[$j$] $\wedge$ *my_color* = green
        *my_color* := red;

A7: **upon** receive *dack(m)* from $M_j$
        **if** ($m \in S[j]$) S[$j$] := $S[j] - \{m\}$;
        **else** $R[j]$ := $R[j] \cup \{m\}$;
        *dacks_rcvd*[$j$]++;
        *dack_pending*[$j$] := false;
        CP[$j$] := *chanp*$_{ij}$(S[$j$]);
```

Figure 6: Monitor Process M_i

3.3.3 Receiving New Snapshots

Each monitor process, M_i, is responsible for labeling snapshots from P_i red, and for maintaining the current state of the channels on which P_i sends application messages. Thus, the global state is advanced in parallel. Whenever monitor process M_i receives a snapshot, it labels its current state green and updates the channel data structures. Each monitor process has direct access to the send activity for the channels it oversees. However, it must communicate with other monitor processes to learn which messages have been received. Figure 7 shows the procedure that M_i uses after receiving a new snapshot from P_i.

Each send event in the incremental history is handled in the analogous manner as with the centralized algorithm (see Figure 5). However, each record contained in *state.increcv* must be sent in a *dack* message to the monitor process responsible for that channel.

Action A7 in Figure 6 gives the steps that will be followed by the recipient of a *dack* message. Collectively, actions A1 and A7 in the distributed algorithm perform the same function as that of Action A1 in the centralized algorithm.

```
update_channels()
    foreach message $m$ sent by $P_i$ to $P_j$ do:
        if ($m \in R[j]$) $R[j]$ := $R[j] - \{m\}$;
        else $S[j]$ := $S[j] \cup \{m\}$;
        CP[$j$] := $chanp_{ij}$(S[$j$]);
    done

    foreach message $m$ in received by $P_i$ from $P_j$ do:
        send dack($m$) to $M_j$;
        dacks_sent[j]++;
    done
```

Figure 7: Monitor Process M_i — update_channels()

3.3.4 Eliminating Inconsistent States

Action A2 is used to label the current *state* red when it happened before some other state in the current cut. We do not use vector clocks in Figure 6. Vector clocks are required if n (the number of application processes over which the global predicate is defined) is less than N (the total number of application processes). In the distributed algorithm, the use of vector clocks necessitates additional messages between monitor processes which carry the latest vector clock from M_i.state. However, when $n = N$, a simpler test for consistency is $\forall i M_i.R[i] = \emptyset$.

3.3.5 Making Progress for Channel Predicates

Actions A3 through A6 are used to label *states* red according to the value of the channel predicates on the current cut. Since M_i evaluates the channels on which P_i sends application messages, it can label its own *state* red after evaluating any of its channel predicates to be F_r. Action A3 performs this task.

Actions A4 through A6 are used to label the receiving process red when a channel predicate has the value F_s. Recall that when a channel predicate is F_s, the receiving process must receive at least one more message in order for the predicate to change value. Thus, in the case that $M_i.CP[j] = F_s$, M_i has determined that $M_j.state$ must be labeled red. However, M_i can not directly access $M_j.state$, and furthermore, there is no assurance that $M_j.state$ has not already been eliminated (or equivalently, a *dack* message is already in route to M_i). Action A4 is used to request more *dack* messages, since the value of the channel predicate can not change until more dack messages are received. Actions A5 and A6 are used to label M_j*.state* red if and only if more *dacks* have been requested than have already been sent.

3.3.6 Correctness of the Distributed Algorithm

This section presents a proof that the distributed GCP algorithm correctly detects the first global cut that satisfies the GCP. The distributed algorithm is similar to the centralized algorithm, and we base our correctness argument on the proof of Theorem 3.4.

Lemma 3.5 *The following is true when all* dacks *have been received:*

$$\begin{aligned} M_i.S[j] &= M_i.state.Sent[j] - M_j.state.Rcvd[i] \\ M_i.R[j] &= M_j.state.Rcvd[i] - M_i.state.Sent[j] \\ M_i.CP[j] &= chanp_{ij}(M_i.state, M_j.state) \end{aligned}$$

Proof: The proof is similar to that for Lemma 3.2 and Lemma 3.3. The only difference is that when M_j receives a new state, the *increcv* records are not immediately added to M_i.R[j] or subtracted from M_i.S[j]. They must be sent in *dack* messages first, hence the precondition that all *dacks* have been received. □

Lemma 3.6 *The following invariant is a consequence of monotonicity (see Lemma 2.2):*

$$M_i.CP[j] = F_r \Rightarrow chanp_{ij}(M_i.state, M_j.state) = F_r$$

Lemma 3.7 $M_i.dacks\_required[j] > M_i.dacks\_sent[j] \Rightarrow chanp_{ji}(M_j.state, M_i.state) = F_s$

Proof: $M_i.dacks\_required[j] > M_i.dacks\_sent[j]$ only if M_i received a *dack_request* message from M_j with *count* $= M_i.dacks\_sent[j] + 1$

Consider the state of M_j at the time when this *dack_request* message was sent. From Action A4, we know that $M_j.dacks\_rcvd[i] = count - 1$. By substitution, $M_j.dacks\_rcvd[i] = M_i.dacks\_sent[j]$. Hence all *dacks* for messages prior to $M_i.state$ were received by M_j prior to the *dack_request* message being sent. Therefore, from Lemma 3.5, $M_j.CP[i] = chanp_{ji}(M_j.state, M_i.state)$. Since the guard for A4 must be true in order for the *dack_request* message to be sent, we know that $chanp_{ji}(M_j.state, M_i.state) = F_s$. □

Theorem 3.8 *The distributed GCP detection algorithm will terminate with* $M_i.state = H[i]$ *iff* H *is the first cut to satisfy the GCP.*

Proof: Initially, $\forall i : M_i.state \prec H[i]$ since each monitor process initializes itself to a fictitious state. As in Theorem 3.4, we show:

1. if $M_i.state = H[i]$ then $M_i.state$ is never labeled red.

2. if $M_i.state \prec H[i]$ then $M_i.state$ is eventually labeled red

3. if $\forall i : M_i.state = H[i]$ then the algorithm will eventually terminate.

At most a finite number (mN) of states can be eliminated, thus the algorithm will always terminate.

Part 1: no state from H is ever labeled red. The proof is by induction on the number of states labeled red so far. Let $M_i.state$ be the next state labeled red. This can happen as a consequence of Actions A2, A3 or A6. Assume that $M_i.state = H[i]$. If the guard for A2 is true, then $\exists j$ such that P_j has received some message before $M_j.state$ that P_i has not sent prior to $M_i.state$. This implies, $M_i.state \rightarrow M_j.state$. By our induction hypothesis, $M_j.state \leq H[j]$, therefore $H[i] \rightarrow H[j]$, a contradiction.

If the guard for A3 were true, then $\exists j$ such that $M_i.CP[j] = F_r$. By Lemma 3.6 we know $chanp_{ij}(M_i.state, M_j.state) =$ F_r. Using a similar argument as used in Theorem 3.4, this leads to $chanp_{ij}(H[i], H[j]) = F_r$, a contradiction.

If the guard for A6 were true, then $\exists j$ such that $M_i.dacks\_sent[j] < M_i.dacks\_required[j]$. By Lemma 3.7, $chanp_{ji}(M_j.state, M_i.state) = F_s$. This leads to $chanp_{ji}(H[j], H[i]) = F_s$, a contradiction.

We thus conclude that $\forall i : M_i.state \leq H[i]$.

Part 2: all predecessors to H[i] are eventually labeled red. The proof is by induction on the number of predecessors to H which must be labeled red. The claim is clearly true when there are zero predecessors to H. Assume that there are k states between the current cut ($\forall i M_i.state$) and H. We show that at least one state is labeled red. There are three cases:

Case 1: $\exists i, j : M_i.state \rightarrow M_j.state$. Since we assume $n = N$, this is equivalent to $\exists i, j : M_i.state \rightsquigarrow M_j.state$. Eventually all *dacks* will be received by M_i. At this point, we know $M_i.R[j] \neq \emptyset$ by the definition of $\rightsquigarrow$. Therefore Action A2 applies, and $M_i.state$ will be labeled red.

Case 2: $\exists i, j : chanp_{ij}(M_i.state, M_j.state) = F_r$. Eventually, all *dacks* will be received. At this point, from Lemma 3.5 we know $M_i.CP[j] = F_r$. Hence, Action A3 applies and $M_i.state$ will be labeled red.

Case 3: $\exists i, j : chanp_{ij}(M_i.state, M_j.state) = F_s$. Eventually, all *dacks* will be received. At this point, we know $M_i.CP[j] = F_s$. Action A4 will cause a *dack_request* message to be sent to M_j. Eventually this message will be received. At this time, we know that Action A5 will set $M_j.dacks\_required[i]$ to be one more than $M_j.dacks\_sent[i]$ (since all *dacks* had been received before the *dack_request* message had been sent). Action A6 will apply, and M_j.state will be labeled red.

We therefore conclude that all predecessors to H are labeled red.

We now conclude the proof by showing that when $\forall i : M_i.state = H[i]$, termination occurs. This fact is clearly seen by noting that Action A1 can be taken at most m times on each M_i since there are at most m snapshots from each process. Actions A2, A3 and A6 can also apply at most m times, since each of these actions causes the state to be labeled red. Each message that is received by P_i causes M_i to send at most one *dack* message. Therefore, Action A7 can apply at most m times. Action A4 can apply at most m times, since at least one *dack* must be received for each *dack_request* message that is sent. And finally, Action A5 can only occur m times, since A4 occurs at most m times.

Therefore, after each M_i has taken $O(m)$ actions, the algorithm will terminate. If H exists, then $\forall i : M_i.state = H[i]$. If H does not exists, then all of the states have been eliminated from at least one process. $\Box$

3.3.7 Overhead Analysis

The distributed algorithm operates using the same principles as the centralized algorithm. The two versions of the algorithms have identical worst case asymptotic time, space and message complexity.

We consider first the number of messages exchanged. We describe the case where $n = N$. Both the centralized and distributed algorithms send mn local snapshots. However, The distributed algorithm requires *dack* and *dack_request* messages which are not needed in the centralized algorithm. Up to mn of each type of message are required. To detect termination, we must double the number of monitor messages. Hence, the distributed algorithm requires $5mn$ messages, whereas the centralized algorithm requires only mn. However, this analysis is somewhat misleading. Recall that M_i and P_i can be located on the same physical processor in the distributed algorithm. Hence no network traffic is generated for sending local snapshots in this case. Furthermore, monitor messages are small. Two 32-bit integers is sufficient to encode a monitor message in practice. Hence, it is quite possible that the distributed algorithm will actually consume less network bandwidth than the centralized algorithm.

We now consider the design tradeoffs related to concurrency. The centralized algorithm suffers from M_0

acting as a serial bottleneck. This can be a significant drawback, particularly if n is very large. The distributed algorithm is able to exploit concurrency. The memory requirements are also evenly distributed over the n processors in the system. Although this appears to indicate a clear win for the distributed algorithm, there are two issues. First, under pathological conditions there may be little or no parallelism available for the distributed algorithm to exploit. In these cases, the distributed algorithm proceeds with only one monitor process being active at a time. Second, the centralized algorithm may have lower detection latency. If H is the first cut to satisfy a GCP, then the detection latency is defined as the wall-clock time between when the last application process reaches H and when the first monitor process detects the GCP. Typically, M_0 will be able to immediately detect the GCP after the last local snapshot is received. In the distributed algorithm the last snapshot may generate several *dack* messages, each of which must be received before the GCP can be detected.

4 Conclusions

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 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 two efficient algorithms to detect the first consistent cut in which a GCP is true. The overhead of our algorithms are bounded by low-order polynomial functions of the number of processes and the number of messages. For many interesting problems, the channel state can be encoded by a simple counter. In these cases the time, space and message complexity of our algorithms are linear in the number of local states.

References

[BM94] O. Babaoğlu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In Sape Mullender, editor, *Distributed Systems*, pages 55–96. Addison Wesley, New York, NY, 2nd edition, 1994.

[CL85] K. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, pages 63–75, February 1985.

[CM91] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163–173, Santa Cruz, California, May 1991.

[DS80] Edsger W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

[Fid89] C. J. Fidge. Partial orders for parallel debugging. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, volume 24 of *SIGPLAN Notices*, pages 183–194, January 1989.

[FRGT94] E. Fromentin, M. Raynal, V. K. Garg, and A. Tomlinson. On the fly testing of regular patterns in distributed computations. In *Proceedings of the 23rd Int. Conference on Parallel Processing*, 1994 1994.

[GW92] V. K. Garg and B. Waldecker. Detection of unstable predicates in distributed programs. In *Proc. 12th Conference on the Foundations of Software Technology Theoretical Computer Science*, Lecture Notes in Computer Science, pages 253–264, New Delhi, India, December 1992. Springer-Verlag.

[GW94] V. K. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, March 1994.

[HW88] D. Haban and W. Weigel. Global events and global breakpoints in distributed systems. In *Proc. of the 21st Intl Conf. on System Sciences*, pages 166–175, 1988.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V, 1989.

[MC88] B. P. Miller and J. Choi. Breakpoints and halting in distributed programs. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 316–323, San Jose, California, June 1988.

[MI92] Y. Manabe and M. Imase. Global conditions in debugging distributed programs. *Journal of Parallel and Distributed Computing*, 15:62–69, 1992.

[SM94] R. Scwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[TG93] A.I. Tomlinson and V. K. Garg. Detecting relational global predicates in distributed systems. In *Proc. 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 21–31, San Diego, California, May 1993.