Deriving Distributed Algorithms from a General Predicate Detector

J. Roger Mitchell* Vijay K. Garg†
Parallel and Distributed Systems Lab
http://maple.ece.utexas.edu
Electrical and Computer Engineering Department
The University of Texas at Austin,
Austin, TX 78712

Abstract

Designing and debugging distributed systems requires the detection of conditions across the entire system. As an illustration, monitoring the status of an application requires detection of termination, and using virtual time requires the periodic calculation of the global virtual time. The Generalized Conjunctive Predicate (GCP) detector offers a method to derive detection algorithms for these and other problems based on optimizing the base algorithm.

1 Introduction

The problem of detecting global conditions arises in designing, testing, and debugging of distributed systems. There have been many approaches to detecting global conditions, or predicates, as in [1, 2, 3, 6, 9, 10, 12, 13, 17]. Some of these approaches can detect conditions within channels as well as at the processes of the system. However, these approaches either cannot perform detection during the computation or cannot detect a very broad range of predicates. The Generalized Conjunctive Predicate (GCP) detector algorithm, introduced in [5, 4], can be used to detect the conjunction of local and channel predicates during the distributed computation. The approach used by the GCP method is to collect states for which the local process predicates are true, and then attempt to find a group of these states which are part of a consistent cut. Channel predicates are checked following this.

We will show that the GCP algorithm can be thought of as a generic algorithm from which other algorithms can be derived. This is done by optimizing GCP according to the properties of detection problems. We will define these properties and the optimizations for the GCP algorithm that stem from them. Then we will demonstrate how to derive algorithms using these optimizations by applying this approach to two specific detection problems, namely: termination detection and channel buffer overflow.

*supported in part by MCD and Virginia & Ernest Cockrell fellowships

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

2 Previous work

Snapshot algorithms[1, 8] attempt to find consistent cuts and then determine if the cuts satisfy the predicate. Because of this, they are suitable only for stable predicates. The GCP algorithm, however, can detect stable as well as unstable predicates. GCP also finds the earliest occurrence of such a predicate, that is, the first consistent cut which satisfies the predicate. Snapshot algorithms cannot, in general, find the first occurrence of a predicate. Table 1 compares the features and complexities of various detection algorithms. CL[1] and M[10] are snapshot algorithms. FRGT[3] and MC[13] detect the class of *linked* predicates. MI[12] uses a replay method. GCKM[5] is the GCP method.

3 The token-GCP algorithm

3.1 Notation

In this paper, we use notation of states, events, and predicates to illustrate the concepts of GCP. A lower case letter is used to represent a process or a channel state and an upper case letter represents a global state, that is, a collection of one local state from every process and the corresponding channel states between these processes. For example, s, t, and u are local process states, while G and H are global states or cuts. We use $LP(x), CP(y)$, and $LP(Z)$ to indicate that a local predicate is true at state x, a channel predicate is true for channel state y, and a local predicate is true for all local states that make up global state Z. For channel predicates, the channel state is a sequence of messages, belonging to any set Σ^* (a sequence of messages). Therefore, $CP : \Sigma^* \rightarrow \{true, false\}$, that is, CP maps a sequence of messages, the channel state, to *true* or *false*.

The ordering of local states are represented using $\prec$, and $\rightarrow$. If $s \prec t$ then s occurs before t on the same process. The relation $s \rightarrow t$ holds if and only if one of the following holds: 1) $s \prec t$, 2) the action following s is the send of a message and the action preceding t is the reception of that message, 3) there exists a state u such that $s \rightarrow u$ and $u \rightarrow t$. Note that when $\prec$ is used as a relation for cuts, as in $G \prec H$, this means that each local state in G is equal to or $\prec$ the state

	Predicates detected	# of msgs	space reqmnts /process	detect latency	msg order	control tag/msg overhead	algorithm complexity	approach
CL[1]	stable l & c	$O(rn)$	$O(m)$	αr	FIFO	$O(1)$	$O(rn)$	centralized
CM[2]	unstable l & c	$O(k^n)$	$O(k^n)$	0	none	$O(1)$	$O(k^n)$	centralized
FRGT[3]	partially ordered l	0	$O(p)$	0	none	$O(p)$	$O(pm)$	decentralized
GW[6]	unstable conj l	$O(mn)$	$O(m)$	0	none*	$O(n)$	$O(nm^2)$	centralized
M[10]	stable l & c	$O(rn)$	$O(m)$	αr	none	$O(1)$	$O(rn)$	centralized
MI[12]	unstable l & c	0	$O(m)$	αt	FIFO	$O(n)$	$O(mn^2)$**	centralized
MC[13]	linked l	$O(pn^2)$	$O(p)$	0	none	$O(p)$	$O(pn^2)$	centralized
TG[17]	relational l	$O(m)$	$O(m)$	0	none	$O(1)$	$O(m^2)$	decentralized
GC[4]	unstable conj l	$O(mN)$ or $O(mn)$	$O(mn)$	0	none*	$O(1)$ or $O(n)$	$O(mN)$ or $O(mn^2)$	decentralized
GCKM[5]	unstable conj l & c	$O(mn)$	$O(mn)$	0	none*	$O(n)$	$O(mn^2)$	decentralized

l = local predicate t = time of program execution n = # processes within predicate
c = channel predicate p = size of predicate N = # processes in system
r = # runs of snapshot k = maximum # events at a process m = # send/receive events at a process
* FIFO ordering required to central process. ** This algorithm requires replay of execution.

Table 1: Comparison of various predicate detection schemes.

on the same process in H, and that at least one local state in $G \prec$ a state in H.

We use m_i to represent message i. The send and receive events are represented by $m_i.send$ and $m_i.receive$. Also, the process on which s occurs is $s.p$.

3.2 Overview of the algorithm

The GCP algorithm was originally presented in [5] which contains a centralized version of the algorithm. The GCP algorithm in this paper is the token based algorithm found in [4]. Token-GCP uses a token to determine when all local and channel predicates are true concurrently. The token contains a possible global cut in which the desired predicates could be true. This global cut contains only local states for which the local predicates are true. If the global cut is also consistent then all local predicates are true concurrently. If not, the token is forwarded to any process which violates the consistency. Once a consistent cut is found, the channel predicates are checked. If true, then the GCP is detected.

The channel predicates detected are within the class called *monotonic*, and further divided into *send* and *receive monotonic* predicates. A *send-monotonic* channel predicate is one which, if false, cannot be made true by sending more messages on a channel. The predicate "the channel is empty" is an example of a send-monotonic predicate, because sending more messages cannot make this predicate true. A *receive-monotonic* channel predicate cannot be made true by receiving more messages from the channel.

The implementation of GCP is divided into application and monitor portions at each process. This is shown in figure 1. The application process checks for local predicates and the monitor receives the token and checks for a consistent cut and then channel predicates. Figure 2 gives a high level overview showing the application and monitor code. Note that the monitor code is only activated when the token has been received (there is only one token in GCP). Statement GCP1 initializes the algorithm and GCP2-3 add a sent or received message to message lists for the application. GCP4 detects when the local predicate is true since the last send or receive. When this occurs the application process sends the message list, called a candidate, to the monitor, M_i (GCP5). The message lists are then cleared(GCP6). The monitor waits to receive the GCP token and when it does, begins receiving candidates from the application process. GCP7 collects information from the application about the channels until a candidate is received with a clock later than that of the token (GCP8). GCP9 checks the token to see if the new cut is inconsistent because of information from another process. This is done by checking for a token vector clock that is less than the received candidate clock. If so, then the token is forwarded to the monitor of that process. Otherwise, GCP10 evaluates channel predicates associated with process i. If all channel predicates are true, the predicate has been detected and GCP is set to true by GCP11.

3.3 Properties of GCP

Many of GCP's interesting uses are characterized by detection of channel predicates. What follows is a list of properties inherent to different distributed

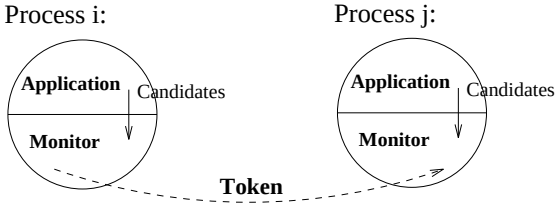

Figure 1: The application and monitor processes of GCP.

Application Process (A_i):

```
Initialize_clocks_and_vars(i);                          GCP1
Before send of message m to A_j do
   Add_send_message_in_order(i, m);                     GCP2
Upon receive of message m from A_j do
   Add_receive_message_in_order(i, m);                  GCP3
Upon local_predicate_true() do {                        GCP4
   send (v, send list, receive list)
      to M_i as candidate                               GCP5
   clear_send_rcv_vars();                               GCP6
   }
```

Monitor Process (M_i):

```
Upon receive of token do {
   do {
      receive candidate from A_i
      Update_send_rcv_of_token(i);                      GCP7
      } until candidate.v[i] > token.v[i]               GCP8
   If ∃j : j ≠ i: token.v[j] < candidate.v[j]
      then send_token(j);                               GCP9
         /* Send to M_j. */
   Else if channel_predicates_true(i)                   GCP10
      then GCP = true.                                  GCP11
   }
```

v = vector clock

Figure 2: Application and monitor code at process i.

programming problems for which the GCP algorithm is suited.

- Channel predicates can be *message content independent*. Those predicates which are independent of message content include those that are based on number or order of messages in a channel.

  The property message content independent is equivalent to:

  $$CP(s) \wedge (|s| = |t|) \Rightarrow CP(t) \quad \text{(R1)}$$

  where CP is the channel predicate and s and t are channel states. So, if (R1) holds, then if the predicate is true at s and the number of messages at t equals that at s, then the predicate is true at t. Termination, deadlock, and buffer overflow are problems with channel predicates that are content independent.

- Channel predicates can be *message ordering independent*. This property is equivalent to:

  $$CP(s) \wedge (s \sim t) \Rightarrow CP(t) \quad \text{(R2)}$$

  where $\sim$ is the relation "is a permutation of." If (R2) holds, then if a predicate is true at a state s, and the messages at s are a permutation of messages at another state, t, then the predicate is true in t as well. Empty channels, and greater than k messages in a channel are predicates which are ordering independent. It should be noted that (R2) represents a weakening of (R1) and therefore, (R1) $\Rightarrow$ (R2). An example of a predicate which satisfies (R2) but not (R1) is "marker in channel".

- Global predicates are either *stable* or *unstable*. These predicates are made up of local and channel predicates. The global predicate, $GP()$, can be written in terms of cuts G and H. Stating that $GP()$ is stable is equivalent to establishing:

  $$GP(H) \wedge (H \prec G) \Rightarrow GP(G) \quad \text{(R3)}$$

- Predicates can be *cut consistency independent*. These predicates can become true in a cut with messages received but not sent. If $G[i]$ represents the state of process i at the cut G, then a consistent cut can be defined as:

  $$\text{consistent\_cut(G)} \equiv \forall i, j : G[i] \not\rightarrow G[j]$$

  Using G and H for cuts, and then consistency independence is equivalent to:

  $$\exists H : CP(H) \wedge LP(H) \iff \exists G : CP(G) \wedge LP(G) \wedge \text{consistent\_cut(G)} \quad \text{(R4)}$$

  (R4) states that a channel and local predicate are true for some cut H, if and only if there exists a consistent cut, G, for which both predicates are true. Termination detection and calculation of global virtual time are consistency independent problems.

- Channel predicates can be *receive monotonic* or *send monotonic*. For example, if a channel predicate is receive monotonic and is true at state s, and the messages in the channel at state s are a prefix of the set of messages at t, then the predicate is also true in t. This can be stated as:

  $$C(s) \wedge s \text{ prefix of } t \Rightarrow C(t) \quad \text{(R5)}$$

  An example of receive monotonicity is the channel predicate "$> k$ messages in channel." Send monotonicity can be written as:

  $$C(s) \wedge t \text{ suffix of } s \Rightarrow C(t) \quad \text{(R6)}$$

  Following the example for receive monotonicity, "$< k$ messages in channel" is a send monotonic channel predicate. As mentioned earlier, the GCP algorithm can only detect monotonic channel predicates.

3.4 Classifying detection problems

Knowledge of properties that make up a detection problem can be exploited for optimizing the GCP algorithm. For example, knowing that a problem is message content independent and that this property implies ordering independence, the GCP algorithm can be optimized so that instead of saving messages sent and received, only the count of messages sent and received is maintained. Table 2 gives the list of properties and the optimizations possible when they are present.

Message content independent	Save only number of messages	GCP2 = Increment_send_count() GCP3 = Increment_receive_count()
Message ordering independent	Send and receive lists not strictly ordered	GCP2 = Concatenate_send_messages GCP3 = Concatenate_receive_messages
Global predicate stable	Do not store previous cuts	GCP7 = receive last candidate
Cut consistency independent	Consistent cuts not required	GCP8 = } until no more Delete GCP9 GCP10 = channel_predicates_true() GCP12 = send token based on channel predicate monotonicity.
Strictly receive monotonic	Check only for receive monotonicity	check for F_s removed from channel_predicates_true() monitor process does not evaluate to F_s
Strictly send monotonic	Check only for send monotonicity	check for F_r removed from channel_predicates_true() monitor process does not evaluate to F_r

Table 2: Optimizations for properties of predicates

The standard GCP algorithm saves messages in any channel to message lists by concatenating. For detection problems which are message ordering independent, any method of appending to the message lists can be used. Generally, a method will be chosen which is most efficient for the condition to be detected.

A look at the table shows that for monotonicity the change to the algorithm is to remove values to which the function channel_predicates_true() can evaluate to. This function determines whether channel predicates are true. Any that are not are set to F_s or F_r indicating that the predicate is false and is currently send-monotonic or receive-monotonic respectively. Predicates that are not strictly send or receive monotonic can evaluate to F_s or F_r indicating that candidates with more sends or receives should be considered.[1]

In this paper, the problems of termination detection and channel buffer overflow are implemented. Other detection problems for which the GCP algorithm is suitable include global virtual time $> k$, system wide deadlock, loss of token, too many marker colors, violation of FIFO, and channel underutilization.

Knowing the properties for a given problem will allow us to optimize the GCP algorithm in each case.

4 GCP for Termination Detection

Termination detection is a standard problem in distributed computing. For this problem, an idle process can be defined as:

> A process, if idle, remains idle until it receives a message. (T1)

Termination detection, then, usually involves checking that the following holds:

> All processes are idle and there are no messages in the channels. (T2)

It is not difficult to see that if (T1) holds and (T2) becomes true then the system is terminated and will remain terminated until some message is sent from outside the system. Because of this, termination is a stable property and satisfies (R3). The channel predicate is also content, order, and can be shown to be cut consistency independent. Obviously, detection of empty channels only requires knowledge of the number of messages in the channel, without concern for their order. So, (R1) is satisfied which implies (R2) as well.

Let s_{ij} and r_{ij} be the number of sends and receives from i to j respectively. Since a channel is empty when the number of sends equals receives on that channel,

$$\bigwedge_{i,j:i\neq j} s_{ij} = r_{ij} \iff \text{all channels empty} \qquad (1)$$

So, to use GCP for termination detection requires that the local and channel predicates be defined as:

ith process's local predicate $\equiv$ process i idle

Channel i, j predicate $\equiv s_{ij} = r_{ij}$

The states where s_{ij} and r_{ij} are measured are on different processes. However, consistency of these states is not a concern. Equation 1 is true independent of whether or not these two variables are measured when processes i and j are concurrent. This is because s_{ij} and r_{ij} are strictly increasing. In fact, when s_{ij} does equal r_{ij}, the two will be concurrent. The GCP algorithm, though, does perform predicate detection using consistent cuts. We will look at optimizing GCP so that consistent cuts are not used.

The GCP optimization would be removal of the check for consistency. There is, however, another improvement that can be made with this approach and it involves an approach first presented by Mattern in his *vector counting* algorithm [11]. Let $G[i]$ represent the state of P_i at the cut and let:

[1] Some predicates are *dynamically monotonic*, that is, these predicates can be send or receive monotonic based upon their evaluation. For example, the predicate "the channel contains six messages" is send-monotonic if there are greater than six messages in the channel and receive-monotonic if there are less than six messages.

$G[i].r$ = # of messages rcvd by P_i at the cut.
$G[j].s[i]$ = # of sent messages from P_j to P_i.
$G[j].R[j]$ = history of messages received at i from j at the cut.
$G[i].S[j]$ = history of messages sent from j to i at the cut.

A theorem can be written that relates the number of receives at a process to the number of send to that process.

Theorem 4.1 *Given (T1) and* $\forall i : G[i] = idle$:
$\forall i, j : G[i].r = \sum_j G[j].s[i] \iff$
$\forall i, j : G[i].R[j] = G[j].S[i]$.
That is, for all i, the number of messages recorded as received by i at the cut equals the sum of sends to i by all other processes, if and only if for all channels, the set of messages sent is equivalent to the set of messages received (or equivalently, the channels are empty).

Proof: See the technical report for this paper or Mattern's paper [11]. □

So, theorem 2.1 says that if $\forall i : G[i].r = \sum_j G[j].s[i]$ is true for any cut, then the channels are empty. In addition, $\forall i, j : G[i].R[j] = G[j].S[i]$ is true, and the cut is consistent. Because of this, the requirement for cut consistency independence, (R4), holds for termination detection, and in this case, $G = H$.

Now termination detection can be implemented with its properties: message content, message ordering, and cut consistency independence. Using the optimization in table 2 to implement the changes associated with these properties gives the algorithm in figure 3. Note that the channel predicate on any channel from process i to process j is $R_j = \sum_k S_{kj}$, as described above. The variables r and s_j, as specified above, are maintained by A_i, while R_j and S_{ij} are contained in the token.

5 GCP for detection of channel buffer overflow

The problem of monitoring channel buffers is to detect when too many messages have been sent without being received. The reason would be to detect a sender that has violated the limit on the maximum number of messages to send before an acknowledgement from the receiver.

Note that the channel predicate is message content and ordering independent. It is also receive monotonic because we are detecting "# messages $> k$". The local predicates used here would be set to TRUE. To use GCP requires that the local and channel predicates be defined as:

ith process's local predicate $\equiv$ TRUE

Channel i, j predicate $\equiv s_{ij} - r_{ij} > k$

where s_{ij} and r_{ij} are send and receive counts. Unlike termination detection, the channel predicate does

Application Process (A_i):

```
Initialize_clocks_and_vars(i);
Before send of message m to A_j do
   Increment_send_count(i, j);
      /* send_count_j ++ */
Upon receive of message m from A_j do
   Increment_receive_count(i, j);
      /* receive_count++ */
Upon local_predicate_true() do {
   /* Is application process idle? */
   send (v, all send_count_j, receive_count) to
      M_i as candidate
   clear_send_rcv_vars();
      /* clear receive_count & all send_count_j . */
   }
```

Monitor Process (M_i):

```
Upon receive of token do {
   do {
      receive candidate from A_i
      Update_send_rcv_of_token(i);
      } until no more candidates;
   If channel_predicates_true(i)
      /* If NOT true, token sent
         to another monitor. */
      then GCP = true.
   else send token based on channel monotonicity;
   }
```

Figure 3: GCP at P_i optimized for termination detection

require a consistent cut. This is because if inconsistency were allowed, a cut may be found in which the number of messages detected in the channel is greater than that of any consistent cut. The inconsistent cut shown in figure 4 detects five messages in the channel from i to j when actually a consistent cut will record at most three.

Because detection of channel buffer overflow is message content and ordering independent, as well as receive monotonic, the optimizations for these can be implemented in the GCP algorithm. Because the local predicate is TRUE, this check is eliminated in the GCP application code so that a **candidate** is sent to

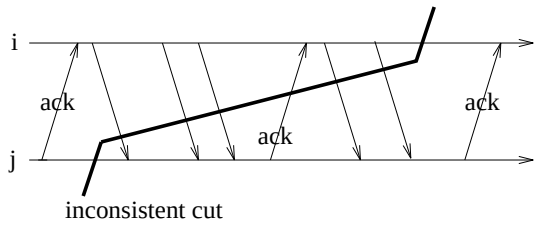

Figure 4: An example of an inconsistent cut which would incorrectly detect > 3 messages.

Application Process (A_i):

```
Initialize_clocks_and_vars(i);
Before send of message m to A_j do
   Increment_send_count(i, j);
      /* send_count_j ++ */
Upon receive of message m from A_j do
   Increment_receive_count(i, j);
      /* receive_count_i ++ */
Upon local_predicate_true() do {
   /* Local predicate always true; check firstflag. */
   send (v, all send_count_j, all receive_count_j) to
      M_i as candidate
   clear_send_rcv_vars();
      /* clear all receive_count_j and send_count_j. */
   }
```

Monitor Process (M_i):

```
Upon receive of token do {
   do {
      receive candidate from A_i
      Update_send_rcv_of_token(i);
      } until candidate.v[i] > token.v[i]
   If ∃j : j ≠ i: token.v[j] < candidate.v[j]
      then send token to M_j.
   Else if channel_predicates_true(i)
      then GCP = true.
   }
```

Figure 5: GCP at process i for channel capacity usage.

the monitor after every send and receive. Implementing these changes gives the algorithm in figure 5.

6 Conclusions

We have given properties of distributed detection problems and shown how to use these properties to optimize the GCP algorithm for distributed systems. Therefore, GCP can be thought of as a base algorithm from which algorithms for many standard problems can be derived. These optimizations can be performed for unstable as well as stable predicates, which allows for a broader range of problems. In addition, the optimized GCP algorithms will detect the first occurrence of any predicate.

References

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

[2] 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.

[3] 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.

[4] V. K. Garg, C. Chase, "Distributed Algorithms for Detecting Conjunctive Predicates," Technical Report ECE-PDS-1994-003, ECE Dept, University of Texas at Austin, 1994.

[5] V. K. Garg, C. Chase, R. Kilgore, J. R. Mitchell ,"Detecting Conjunctive Channel Predicates in a Distributed Programming Environment," *Proceedings of the Hawaii International Conference on System Sciences*, Maui, Hawaii, Vol 2, January 1995, pp. 232-241.

[6] 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.

[7] 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.

[8] T. H. Lai, T. H. Yang, "On Distributed Snapshots," *Global States and Time in Distributed Systems*, pp. 23-26, IEEE Computer Society Press, Los Alamitos, CA 1994.

[9] 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.

[10] 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.

[11] F. Mattern, "Algorithms for distributed termination detection," *Distributed Computing*, 2:161-175, 1987.

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

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.