Detecting Relational Global Predicates in Distributed Systems *

Alexander I. Tomlinson Vijay K. Garg

Department of Electrical and Computer Engineering
The University of Texas at Austin, Austin, Texas 78712

This paper appears in:
Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging,
San Diego, CA, May, 1993.

Abstract

This paper defines relational global predicates and presents efficient algorithms to detect them in a distributed program which uses unordered asynchronous messages for inter-process communication. We use relational global predicates of the form $(x0 + x1 < C)$ where $x0$ and $x1$ are integer values at processes $P0$ and $P1$ in a system of N processes. We present a fully decentralized algorithm that runs concurrently with the target program, uses constant size message tags (four integers), and generates one debug message for each message received by $P0$ and $P1$. We also describe a centralized algorithm that can be used as a checker process which runs concurrently with the target program, or after the target program terminates. We generalize our results to an algebra $(D, \%, *)$ where $\%$ and $*$ are binary operators in domain D, $\%$ is commutative, associative and idempotent, and $*$ distributes over $\%$. In this algebra we can calculate value of the expression $(v1 \,\%\, v2 \,\%...\%\, vn)$ where $\{v1, v2, ...vn\}$ is the set which contains the value of $x0 * x1$ in each consistent cut. For example if $(D, \%, *) = (\text{Integers}, \min, +)$ then we could calculate the minimum value of $x0 + x1$ over all consistent cuts. This generalization opens up many useful variants of our algorithm, including detection of weak conjunctive boolean predicates.

1 Introduction

A condition that depends on the state of multiple processes in a distributed system is called a global predicate. Detection of global predicates is a fundamental problem in distributed computing; it arises in many contexts such as design, testing and debugging of distributed programs. There are two types of global predicates: stable and unstable. A stable predicate is one that remains true once it becomes true. An unstable predicate may alternate between true and false.

Chandy and Lamport [CL85] give an algorithm to detect stable predicates that uses global snapshots. Bouge [Bou87], and Spezialetti and Kearns [SK86] extend this method for repeated snapshots. Spezialetti and Kearns [SK88] discuss methods for recognizing monotonic event occurrences without taking snapshots. These approaches do not work for unstable predicates because the predicate may become true and then false again in between two snapshots. An entirely different approach is required for unstable predicates.

Earlier work shows how to detect unstable predicates which can be expressed as a conjunction, disjunction or sequencing of local predicates. Garg and Waldecker [GW92, GW94] define strong and weak conjunctive predicates and present efficient algorithms for detecting them. Hurfin, Plouzeau and Raynal [HPR93] discuss methods for detecting atomic sequences of predicates.

Cooper and Marzullo [CM91], and Haban and Weigel [HW88] give algorithms for detection of general global predicates. Detection of such predicates is intractable since the number of global states is exponential and each state must be explicitly checked.

In this paper we continue the study of detection of unstable predicates by considering relational global predicates which cannot be decomposed into a conjunction, disjunction or sequencing of local predicates.

*Research supported in part by NSF Grant CCR 9110605, Navy Grant N00039-88-C-0082, TRW faculty assistantship award, IBM Agreement 153, and an MCD University Fellowship.

The predicate $(x_0 + x_1 < C)$ belongs to this class, where x_0 and x_1 are integers in different processes and C is a constant.

Relational global predicates are useful for detecting potential violations of a limited resource. For example, consider a server which can handle at most 5 connections at a time. Client processes P_0 and P_1 each have a variable x_0 and x_1 which indicates the number connections it has with the server. The predicate $(x_0+x_1 > 5)$, which can be reformulated as $((-x_0) + (-x_1) < -5)$, indicates a potential error. Although the ability to detect relational global predicates with N terms would be more useful, it is currently not known how to do this efficiently.

We present two algorithms for detecting relational global predicates. The decentralized algorithm runs concurrently with the target program and can be used for online detection of the predicate. The centralized version is decoupled from the target program and can run concurrently with the target program or post-mortem (i.e., after the target program terminates). We formally prove that both algorithms are sound (if the predicate occurs, then it is detected) and complete (if the predicate is detected, then it has occurred).

2 Model and Notation

We use the following notation for quantified expressions: (Op FreeVars : Range of FreeVars : Expr). Op can be any commutative associative operator (eg, min, $\cup, +$). For example $(\min i : i \in \mathcal{R} : f(i))$ is the minimum value of $f(i)$ for all i such that $i \in \mathcal{R}$.

Any distributed computation can be modeled as a decomposed partially ordered set (deposet) of process states [Gar92, Fid89]. A deposet is a tuple $(S_0, S_1, \ldots S_N, \rightsquigarrow)$ such that:

1. S_i is a finite sequence of local states. We say that $a \triangleright b$ if and only if a immediately precedes b in S_i. Thus, S_i is an irreflexive totally ordered set under transitive closure of $\triangleright$.
2. Let $S = (\cup i : 0 \leq i \leq N : S_i)$ and let $\rightarrow$ be the transitive closure of $\triangleright \cup \rightsquigarrow$. Then $(S, \rightarrow)$ is an irreflexive partial order.

An execution that consists of processes $P_0, P_1, \ldots P_N$ can be modeled by a deposet where S_i is the set of local states at P_i which are sequenced by $\triangleright$; the $\rightsquigarrow$ relation represents the ordering induced by messages; and $\rightarrow$ is Lamport's *happened-before* relation[Lam78].

If $(u \rightarrow v)$ then $max(u, v) = v$ and $min(u, v) = u$. Since max and min are commutative and associative, the maximum and minimum element of any chain in $(S, \rightarrow)$ is well defined. The unit elements of the max and min operators are $\perp$ and $\top$ respectively. Thus max applied to a zero length chain returns $\perp$. We require that $(\forall u : u \in S : \perp \rightarrow u \ \wedge\ u \rightarrow \top)$, and also that $\perp \rightarrow \perp$ and $\top \rightarrow \top$.

The predecessor and successor functions are defined as follows for $u \in S$ and $0 \leq i \leq N$:

$$pred.u.i = (\max v : v \in S_i \ \wedge\ v \rightarrow u : v)$$

$$succ.u.i = (\min v : v \in S_i \ \wedge\ u \rightarrow v : v)$$

Thus if $(pred.u.i = v)$ then v is the maximum element in $(S_i, \rightarrow)$ which happened before u, or $\perp$ if no element in S_i happened before u.

An external event is the sending or receiving of a message. The n^{th} interval in P_i (denoted by (i, n)) is the subchain of $(S_i, \rightarrow)$ between the $(n-1)^{th}$ and n^{th} external events. For a given interval (i, n), if n is out of range then (i, n) refers to $\perp$ or $\top$. The notion of intervals is useful because the relation of two states belonging to the same interval is a congruence with respect to $\rightarrow$. Thus, for any two states s, s' in the same interval and any state u: $(s \rightarrow u \iff s' \rightarrow u)$ and $(u \rightarrow s \iff u \rightarrow s')$. We exploit this congruence in our algorithms by assigning a single timestamp to all states belonging to the same interval.

The concurrency relation on S is defined as $u \parallel v = (u \not\rightarrow v) \ \wedge\ (v \not\rightarrow u)$. Using this relation, a *state cut* is defined to be a set c of N states such that $(\forall u, v : u, v \in c : u \parallel v)$. Note that a state cut must contain exactly one state from each process.

Due to the congruence mentioned above, the *pred* and *succ* functions and the $\parallel$ relation are well defined on intervals. Similarly, the notion of state cuts can be extended to *interval cuts*.

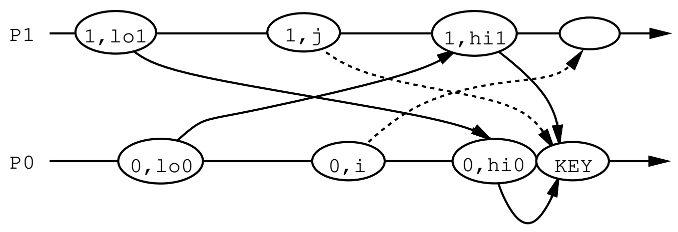

Figure 1: Relationship among intervals when KEY is in P_0. Arrows with solid (dashed) lines represent the $pred$ ($succ$) function.

The value of a variable x in a state $\sigma \in S$ is denoted by $\sigma.x$. The predicate to be detected, previously expressed as $(x_0 + x_1 < C)$, can be stated formally as:

$$(\exists \sigma_0, \sigma_1 : \sigma_0 \in S_0 \ \wedge \ \sigma_1 \in S_1 \ \wedge \ \sigma_0 \parallel \sigma_1 : \sigma_0.x + \sigma_1.x < C)$$

3 Preliminary Results

The following lemma is important in developing efficient algorithms for detecting relational global predicates. It indicates that we need not maintain the value of x in every state, but instead we can maintain the minimum value of x in each interval. We use the notation $minx.(i, m_i)$ to represent the minimum value of x in interval (i, m_i). Formally, $minx.(i, m_i) = (\min \sigma : \sigma \in (i, m_i) : \sigma.x)$.

Lemma 1 $(\exists \sigma_0, \sigma_1 : \sigma_0 \in S_0 \ \wedge \ \sigma_1 \in S_1 \ \wedge \ \sigma_0 \parallel \sigma_1 : \sigma_0.x + \sigma_1.x < C)$
$\iff (\exists m_0, m_1 : (0, m_0) \parallel (1, m_1) : minx.(0, m_0) + minx.(1, m_1) < C)$

Proof: Follows from congruence between intervals and states and from properties of addition over integers. ■

Lemma 2 $(p, i) = pred.(q, j).p \quad \iff \quad (\forall k : k > i : (p, k) \not\rightarrow (q, j))$

Proof:

$$\begin{array}{ll} & (p, i) = pred.(q, j).p \\ \iff & \{\text{definition of } pred\} \\ & (p, i) = (\max k : (p, k) \rightarrow (q, j) : (p, k)) \\ \iff & \\ & (\forall k : k > i : (p, k) \not\rightarrow (q, j)) \end{array}$$

■

Lemma 3 $(p, i) = succ.(q, j).p \quad \iff \quad (\forall k : k < i : (q, j) \not\rightarrow (p, k))$

Proof: Similar to the proof of lemma 2. ■

Lemmas 2 and 3 are used in the proof of lemma 4, which provides the insight needed to understand the algorithms for detecting the relational global predicate. It provides a mechanism for monitoring all intervals in P_0 and P_1 which are concurrent. The lemma states that two intervals $(0, i)$ and $(1, j)$ are concurrent if and only if there exists a sequence of intervals in P_0 which includes $(0, i)$, and a sequence of intervals in P_1 which includes $(1, j)$ such that every interval in the sequence at P_0 is concurrent with every interval in the sequence at P_1. Note that these are sequences of intervals, which themselves are sequences of states. See figure 1 for a graphical representation of the case (in the proof) where KEY is in P_0.

Lemma 4 *Two intervals* $(0, i)$ *and* $(1, j)$ *are concurrent* if and only if *there exist an interval* KEY *in process* P_0 *or* P_1 *such that* $lo_0 < i \leq hi_0$ *and* $lo_1 < j \leq hi_1$*, where*

$$\begin{array}{lcl} (0, hi_0) & := & pred.KEY.0 \\ (1, hi_1) & := & pred.KEY.1 \\ (0, lo_0) & := & pred.(1, hi_1).0 \\ (1, lo_1) & := & pred.(0, hi_0).1 \end{array}$$

Proof: Proof of $\Rightarrow$:
Assume $(0,i) \parallel (1,j)$; thus $(0,i) \not\rightarrow (1,j)$ and $(1,j) \not\rightarrow (0,i)$.
Case A: $succ.(0,i).1 \not\rightarrow succ.(1,j).0$.
Let $KEY = succ.(1,j).0$; then $(1,j) \rightarrow KEY$. Since KEY is in P_0 and $(0, hi_0) = pred.KEY.0$, then $KEY = (0, hi_0 + 1)$. Thus $(1,j) \rightarrow (0, hi_0 + 1)$.

1. $(1,j) \rightarrow (0, hi_0 + 1) \ \wedge \ (1,j) \not\rightarrow (0,i)$
$\Rightarrow$ { transitivity of $\rightarrow$ }
$(0, hi_0 + 1) \not\rightarrow (0,i) \ \wedge \ i \neq hi_0 + 1$
$\Rightarrow$
$i \leq hi_0$

2. $(1, hi_1) = pred.KEY.1 \ \wedge \ (1,j) \rightarrow KEY$
$\Rightarrow$ { lemma 2 }
$(\forall k : k > hi_1 : (1,k) \not\rightarrow KEY) \ \wedge \ (1,j) \rightarrow KEY$
$\Rightarrow$
$(1, hi_1 + 1) \not\rightarrow KEY \ \wedge \ (1,j) \rightarrow KEY$
$\Rightarrow$ { transitivity of $\rightarrow$ }
$(1, hi_1 + 1) \not\rightarrow (1,j) \ \wedge \ j \neq hi_1 + 1$
$\Rightarrow$
$j \leq hi_1$

3. $(1, lo_1) = pred.(0, hi_0).1 \ \wedge \ (0, hi_0 + 1) = succ.(1,j).0$
$\Rightarrow$ { definition of pred, and lemma 3 }
$(1, lo_1) \rightarrow (0, hi_0) \ \wedge \ (\forall k : k < hi_0 + 1 : (1,j) \not\rightarrow (0,k))$
$\Rightarrow$
$(1, lo_1) \rightarrow (0, hi_0) \ \wedge \ (1,j) \not\rightarrow (0, hi_0)$
$\Rightarrow$ { transitivity of $\rightarrow$ }
$(1,j) \not\rightarrow (1, lo_1) \ \wedge \ lo_1 \neq j$
$\Rightarrow$
$lo_1 < j$

4. $succ.(0,i).1 \not\rightarrow KEY \ \wedge \ (1, hi_1) = pred.KEY.1 \ \wedge \ (0, lo_0) = pred.(1, hi_1).0$
$\Rightarrow$ { definition of pred, and let $(1,s) = succ.(0,i).1$ }
$(1,s) \not\rightarrow KEY \ \wedge \ (1, hi_1) \rightarrow KEY \ \wedge \ (0, lo_0) \rightarrow (1, hi_1)$
$\Rightarrow$ { transitivity of $\rightarrow$ }
$(1,s) \not\rightarrow (1, hi_1) \ \wedge \ (1,s) \neq (1, hi_1) \ \wedge \ (0, lo_0) \rightarrow (1, hi_1)$
$\Rightarrow$ { simplify, and apply lemma 3 to $(1,s) = succ.(0,i).1$ }
$hi_1 < s \ \wedge \ (\forall k : k < s : (0,i) \not\rightarrow (1,k)) \ \wedge \ (0, lo_0) \rightarrow (1, hi_1)$
$\Rightarrow$
$(0,i) \not\rightarrow (1, hi_1) \ \wedge \ (0, lo_0) \rightarrow (1, hi_1)$
$\Rightarrow$ { transitivity of $\rightarrow$ }
$(0,i) \not\rightarrow (0, lo_0) \ \wedge \ i \neq lo_0$
$\Rightarrow$
$lo_0 < i$

Case B: $succ.(0,i).1 \rightarrow succ.(1,j).0$.
The proof for case B is abbreviated due to its similarity with case A. Let $KEY = succ.(0,i).1$.

1. $(0,i) \rightarrow (1, hi_1 + 1) \ \wedge \ (0,i) \not\rightarrow (1,j) \quad \Rightarrow \quad j \leq hi_0$
2. $(0, hi_0) = pred.KEY.0 \ \wedge \ (0,i) \rightarrow KEY \quad \Rightarrow \quad i \leq hi_0$
3. $(0, lo_0) = pred.(1, hi_1).0 \ \wedge \ (1, hi_1 + 1) = succ.(0,i).1 \quad \Rightarrow \quad lo_0 < i$
4. $(0, hi_0) = pred.KEY.1 \ \wedge \ KEY \rightarrow succ.(1,j).0 \ \wedge \ (1, lo_1) = pred.(0, hi_0).1 \quad \Rightarrow \quad lo_1 < j$

End proof of $\Rightarrow$.

Proof of $\Leftarrow$:

$$\begin{array}{ll}
1. & (0, lo_0) = pred.(1, hi_1).0 \ \wedge \ lo_0 < i \ \wedge \ j \leq hi_1 \\
\Rightarrow & \{ \text{ lemma 2 } \} \\
 & (\forall k : k > lo_0 : (0,k) \not\rightarrow (1, hi_1)) \ \wedge \ lo_0 < i \ \wedge \ j \leq hi_1 \\
\Rightarrow & \\
 & (0,i) \not\rightarrow (1, hi_1) \ \wedge \ j \leq hi_1 \\
\Rightarrow & \\
 & (0,i) \not\rightarrow (1,j)
\end{array}$$

2. Similarly, $(1,j) \not\rightarrow (0,i)$. Thus $(0,i) \parallel (1,j)$.

End proof of $\Leftarrow$. ■

A corollary to lemma 4 is that the KEY interval is initiated by a message receipt. Intervals that are initiated by a message receipt will be referred to as *receive intervals*, and receive intervals that satisfy the requirements of KEY in lemma 4 will be referred to as *key intervals*.

Corollary 1 *If an interval is a key interval, then it is also a receive interval.*

Proof: Proof follows from the fact that KEY is always defined to be the successor of some interval on another process. If the previous external event was not a message receive, then KEY could not be a successor to any interval on another process. ■

4 Overview of Algorithms

The centralized and the decentralized algorithms gather the same information as the target program executes. They differ in what they do with the information. This section explains what the information is, and how it is gathered. Everything in this section applies to both algorithms.

Each process P_i, for $i \in \{0, 1\}$, must be able to evaluate $minx.(i,m)$ for each interval (i,m). Implementation of the $minx$ function is straight forward. Additionally, for the algorithms to be complete, it is required that: 1) P_1 sends a `<FINAL>` message to P_0 immediately before P_1 terminates, 2) delivery of this message is delayed until immediately before P_0 terminates, and 3) P_0 must receive the message before terminating. This triggers the algorithm at P_0 and ensures that detection is complete. In order for these requirements to be met, both P_0 and P_1 must eventually terminate. This requirement can be removed by adding an additional requirement that P_1 periodically send a message to P_0 to trigger the algorithm.

The algorithms are based on lemma 4. Each message received at P_0 or P_1 begins a receive interval which is a potential key interval (i.e., satisfies lemma 4). Each key interval defines values for lo_0, hi_0, lo_1, and hi_1. The values of lo_0 and hi_0 define a sequence of intervals at P_0. Every interval in this sequence is concurrent with every interval in the corresponding sequence at P_1. Since lemma 4 uses an *if and only if* relation, every pair of concurrent intervals at P_0 and P_1 will appear in the sequences that result from receive interval. Thus if both P_0 and P_1 check the predicate for all pairs of states in these sequences each time a message is received, then the predicate is detected soundly and completely.

We must show how to determine values for lo_0, hi_0, lo_1, and hi_1 in each receive interval. From lemma 4 it can be seen that for any receive interval KEY:

$$\begin{array}{rclcrcl}
(0, hi_0) & = & pred.KEY.0 & \qquad & (0, lo_0) & = & pred.(pred.KEY.1).0 \\
(1, hi_1) & = & pred.KEY.1 & \qquad & (1, lo_1) & = & pred.(pred.KEY.0).1
\end{array}$$

This information can be obtained by using a 2×2 matrix clock as described by Raynal [Ray92]. Let M_k denote the matrix clock at P_k. The following description applies to an $N \times N$ matrix clock in a system with N processes. The 2×2 matrix is the upper left submatrix of the $N \times N$ matrix.

Row k of M_k (i.e. $M_k[k, \cdot]$) is equivalent to a traditional vector clock [Mat89, Fid89]. (In the following explanation, the phrase "P_i's vector clock" refers to row i of P_i's matrix clock.) $M_k[k,k]$ is the local clock of P_k which is incremented at the beginning of each interval. Therefore the current interval is given by $(k, M_k[k,k])$. Since the vector clock implements the $pred$ function, row k of M_k can be used to find

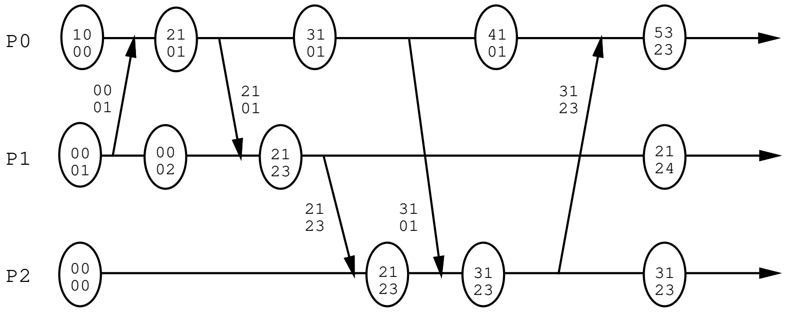

Figure 2: Matrix clock example: Note that for $0 \leq i \leq 1$, row i of P_i's matrix is a traditional vector clock restricted to indices 0 and 1, and row $(i+1)$ *mod* 2 equals the value of $P_{(i+1)\ mod\ 2}$'s vector clock at an interval in the "past" of P_i.

predecessors of $(k, M_k[k,k])$. Furthermore, row k of M_k equals the diagonal of M_k, and row $i \neq k$ is the value of P_i's vector clock when P_i was in interval $(i, M_k[i,i])$. Therefore, row $i \neq k$ can be used to find $pred.pred.()$ of the current interval at P_k. The meaning of a matrix clock is formally stated below, and figure 2 shows values of the matrix clock and message tags on an example run.

- The current interval is given by $(k, M_k[k,k])$
- $(\forall i,j : i \neq j : (j, M_k[i,j]) = pred.(i, M_k[i,i]).j)$
- $(\forall i :: M_k[i,i] = M_k[k,i])$

We can now determine how to evaluate lo_0, hi_0, lo_1, and hi_1. Let $(0,n)$ be any interval in P_0 and M_0 be the value of P_0's matrix clock in interval $(0,n)$:

$$\begin{array}{lclclcl}
(0, hi_0) & = & pred.(0,n).0 & = & (0, n-1) & = & (0, M_0[0,0]-1) \\
(1, hi_1) & = & pred.(0,n).1 & = & pred.(0, M_0[0,0]).1 & = & (1, M_0[0,1]) \\
(0, lo_0) & = & pred.(1, M_0[0,1]).0 & = & pred.(1, M_0[1,1]).0 & = & (0, M_0[1,0]) \\
(1, lo_1) & = & pred.(0, n-1).1 & = & pred.(0, M_0^{n-1}[0,0]).1 & = & (1, M_0^{n-1}[0,1])
\end{array}$$

where M_0^{n-1} is the value of M_0 in the previous interval $(0, n-1)$. Expressions for the values of lo_0, hi_0, lo_1, and hi_1 in P_1 can be determined similarly.

The algorithm for maintaining the matrix clock is presented below. The algorithm is easier to understand by noticing the vector clock algorithm embedded within it: If the row index is held constant, then it reduces to the vector clock algorithm.

<u>Maintaining $M_k[0..1, 0..1]$ at P_k, $0 \leq k \leq N$</u>

To initialize:
  $M_k[\cdot,\cdot] := 0$;
  if $(k = 0 \vee k = 1)$ then $M_k[k,k]++$

To send a message:
  tag message with $M_k[\cdot,\cdot]$;
  if $(k = 0 \vee k = 1)$ then $M_k[k,k]++$ *increment local clock*

Upon receipt of a message tagged with $W[\cdot,\cdot]$:
  for $i := 0$ to 1 do
    if $(M_k[i,i] < W[i,i])$ then $M_k[i,\cdot] := W[i,\cdot]$
  if $(k = 0 \vee k = 1)$ then
    $M_k[k,k]++$ *increment local clock*
    $M_k[k,\cdot] := diagonal(M_k)$

We have shown how to determine the values of lo_0, hi_0, lo_1, and hi_1 in any interval at P_0 or P_1. Thus in the remainder of the paper we refer directly to lo_0, hi_0, lo_1, and hi_1 instead of referring to the vector clock or the *pred* function.

5 Decentralized Algorithm

We describe the algorithm from P_0's point of view. The algorithm at P_1 is symmetrical. Each time a message is received we evaluate lo_0, lo_1, hi_0, and hi_1 as shown in section 4. These values define a sequence of intervals at P_0 and at P_1. The sequence at P_i starts at $(i, lo_i + 1)$ and ends at (i, hi_i). By lemma 4, every interval in the sequence at P_0 is concurrent with every interval at P_1. Thus we can find the minimum value of $minx.(0, i)$ over all intervals $(0, i)$ in the sequence at P_0; call this value min_x_0, and similarly for min_x_1. If the sum of these two values is less than C, then the predicate occurred. Furthermore, since lemma 4 is stated with the *if and only if* relation, if the predicate occurs then this method will detect it.

$$min\_x_0 = (\ \min i\ :\ lo_0 < i \le hi_0\ :\ minx.(0, i)\ )$$
$$min\_x_1 = (\ \min j\ :\ lo_1 < j \le hi_1\ :\ minx.(1, j)\ )$$
if $(min\_x_0 + min\_x_1 < C)$ then PREDICATE_DETECTED

To implement this, min_x_0 can be computed locally. Then P_0 can send a message to P_1 containing (min_x_0, lo_1, hi_1), and P_1 can finish the calculation. This message is a *debug message* and is not considered an external event (i.e., does not initiate a new interval). Messages that are not debug messages are *application messages*.

5.1 Overhead and Complexity

Let M_i be the number of intervals at P_i (also equals the number of application messages sent and received by P_i), and let R_i be the number of receive intervals at P_i (also equals the number of application messages received by P_i).

The message overhead consists of the number and size of the debug messages, and the size of message tags on application messages. P_0 sends one debug message to P_1 in each of the R_0 receive intervals at P_0. Similarly, P_1 generates R_1 debug messages. Thus the total number of debug messages generated by the decentralized algorithm is $R_0 + R_1$. The size of each debug message is 3 integers. Each application message carries a tag of 4 integers. The debug messages can be combined to reduce message overhead, however this will increase the delay between the occurrence of the predicate and its detection.

The memory overhead in P_0 arises from the need to maintain $minx.(0, \cdot)$ for each of M_0 intervals. This can be reduced (for the average case) by a smart implementation since the elements of the array are accessed in order (i.e., the lower elements can be discarded as the computation proceeds). Likewise, the memory overhead for P_1 is M_1. Other process incur only the overhead needed to maintain the matrix clock (i.e., 4 integers).

The computation overhead at P_0 consists of monitoring the local variable which appears in the relational global predicate, and evaluation of the expression $(\min i : lo_0 < i \le hi_0 : minx.(0, i))$ for each debug message sent (R_0) and received (R_1). The aggregate complexity of this is at most $M_0(R_0 + R_1)$ since there are M_0 elements in $minx.(0, \cdot)$. P_1 has similar overhead. Other processes have neither the overhead of monitoring local variables or computing the expression.

6 Centralized Algorithm

This version of the algorithm can be used as a checker process which runs concurrently with the target program, or which runs post-mortem. We describe the post-mortem version which reads data from trace files generated by P_0 and P_1. Since the trace files are accessed sequentially, the algorithm can be easily adapted to run concurrently with the target program by replacing file I/O with message I/O. First we explain what data is stored in the trace files, then we show how the predicate can be detected by one process which has access to both trace files.

Let R_0 be the number of receive intervals in P_0 and Let $Q0[k]$, $1 \leq k \leq R_0$, be a record containing the values of lo_0,hi_0,lo_1, and hi_1 in the k^{th} receive interval. Define R_1 and $Q1[1 \ldots R_1]$ similarly. The elements of both $Q0$ and $Q1$ must be checked to determine if one of the elements represents a key interval (i.e., satisfies the requirements of KEY in lemma 4.

Both $Q0$ and $Q1$ are sorted in terms of all their fields. That is, for $Q = Q0$ or $Q = Q1$, and for every component $x \in \{lo_0, lo_1, hi_0, hi_1\}$, $Q[\cdot].x$ is sorted. This results from the fact that the elements are generated in order on a single process; thus the receive interval represented by $Q[k]$ "happened before" $Q[k+1]$.

P_0's trace file contains two arrays of data: $minx.(0,i)$ for each interval $(0,i)$, and $Q0[1..R_0]$. Likewise P_1's trace file contains $minx.(1,j)$ and $Q1[1..R_1]$. In section 4 we demonstrated how to evaluate lo_0,hi_0,lo_1 and hi_1 for any interval in P_0 or P_1. Generating the values for the $minx$ function is straight forward.

The trace files are analyzed in two independent passes. We describe a function $check(Q[1 \ldots R])$ such that the predicate has occurred if and only if $check(Q0[1 \ldots R_0])$ or $check(Q1[1 \ldots R_1])$ returns *true*. *Check* uses two heaps: $heap_0$ and $heap_1$. $Heap_p$ contains tuples of the form $\langle n, minx.(p,n)\rangle$ where (p,n) is an interval in P_p. The first element of a tuple h is accessed via $h.interval$; the second element is accessed via $h.value$. The heap is sorted with the *value* field.

The algorithm maintains the following properties (HEAP holds at all times. I1 and I2 hold between statements S4 and S5. k is a program variable):

$$\begin{aligned} HEAP &\equiv (\forall h, p : h \in heap_p : heap_p.top().value \leq h.value) \\ I1 &\equiv (\forall i, p : Q[k].lo_p < i \leq Q[k].hi_p : \langle i, minx.(p,i)\rangle \in heap_p) \\ I2 &\equiv (\forall p :: Q[k].lo_p < heap_p.top().interval \leq Q[k].hi_p) \end{aligned}$$

HEAP is an inherent property of heaps: the top element, heap.top(), is the minimum element in the heap. Heaps are designed to efficiently maintain the minimum element of an ordered set. Statements S1 and S2 ensure I1, which states that in the k^{th} iteration of the for loop, all intervals in the sequences defined by $Q[k]$ are represented in the heaps. Statements S3 and S4 ensure I2, which states that the top of $heap_p$ is in the sequence defined by $Q[k]$. The text of the *check* function is shown below.

```
function check(Q[1...R])
          n0 := 1;  n1 := 1;
          for k := 1 to R {
S1:           while (n0 <= Q[k].hi0)
                  heap0.insert( <n0, minx.(0, n0)> );   n0 := n0 + 1;
S2:           while (n1 <= Q[k].hi1)
                  heap1.insert( <n1, minx.(1, n1)> );   n1 := n1 + 1;
S3:           while (heap0.top().interval <= Q[k].lo0)
                  heap0.removetop();
S4:           while (heap1.top().interval <= Q[k].lo1)
                  heap1.removetop();
S5:           if (heap0.top().value + heap1.top().value < C)
                  return TRUE;
          }
          return FALSE;
```

The following lemma proves the correctness of this algorithm.

Lemma 5 *There exists a value for program variable k such that at statement S5* $(heap_0.top().value + heap_1.top().value < C)$ if and only if $(\exists \sigma_0, \sigma_1 : \sigma_0 \in S_0 \ \wedge\ \sigma_1 \in S_1 \ \wedge\ \sigma_0 \parallel \sigma_1 : \sigma_0.x + \sigma_1.x < C)$

Proof:
Proof of $\Rightarrow$: Let $\langle i, minx.(0,i)\rangle = heap_0.top()$ and $\langle j, minx.(1,j)\rangle = heap_1.top()$.

$$
\begin{array}{ll}
 & heap_0.top().value + heap_1.top().value < C \\
\Rightarrow & \{ \text{ simplify and I2 } \} \\
 & minx.(0,i) + minx.(1,j) < C \ \wedge \ Q[k].lo_0 < i \leq Q[k].hi_0 \ \wedge \ Q[k].lo_1 < j \leq Q[k].hi_1 \\
\Rightarrow & \{ \text{ lemma 4 } \} \\
 & minx.(0,i) + minx.(1,j) < C \ \wedge \ (0,i) \parallel (1,j) \\
\Rightarrow & \{ \text{ lemma 1 } \} \\
 & (\exists \sigma_0, \sigma_1 : \sigma_0 \in S_0 \ \wedge \ \sigma_1 \in S_1 \ \wedge \ \sigma_0 \parallel \sigma_1 : \sigma_0.x + \sigma_1.x < C)
\end{array}
$$

Proof of $\Leftarrow$:

$$
\begin{array}{ll}
 & (\exists \sigma_0, \sigma_1 : \sigma_0 \in S_0 \ \wedge \ \sigma_1 \in S_1 \ \wedge \ \sigma_0 \parallel \sigma_1 : \sigma_0.x + \sigma_1.x < C) \\
\Rightarrow & \{ \text{ Lemma 1 } \} \\
 & (\exists i, j : (0,i) \parallel (1,j) : minx.(0,i) + minx.(1,j) < C) \\
\Rightarrow & \left\{ \begin{array}{l} \text{Referring to lemma 4, if } KEY \text{ is in } P_0 \text{ then let } Q = Q0 \text{ and let } k \text{ equal the number of} \\ \text{messages received at } P_0 \text{ before } KEY\text{. Likewise for } KEY \text{ in } P_1\text{. Then } Q[k] \text{ corresponds to} \\ KEY. \end{array} \right\} \\
 & Q[k].lo_0 < i \leq Q[k].hi_0 \ \wedge \ Q[k].l0_1 < j \leq Q[k].hi_1 \ \wedge \ minx.(0,i) + minx.(1,j) < C \\
\Rightarrow & \{ \text{ Invariant I1 } \} \\
 & \langle i, minx.(0,i)\rangle \in heap_0 \ \wedge \ \langle j, minx.(1,j)\rangle \in heap_0 \ \wedge \ minx.(0,i) + minx.(1,j) < C) \\
\Rightarrow & \{ \text{ Invariant HEAP } \} \\
 & heap_0.top().value \leq minx.(0,i) \ \wedge \ heap_1.top().value \leq minx.(1,j) \ \wedge \ minx.(0,i) + minx.(1,j) < C \\
\Rightarrow & \\
 & heap_0.top().value + heap_1.top().value < C
\end{array}
$$

■

6.1 Overhead and Complexity

If we consider each record written to a trace file to be a debug message then the message complexity analysis is identical to the decentralized algorithm (except that the debug messages have a different destination).

P_0 and P_1 do not need to maintain $minx.(\cdot,\cdot)$, thus the only memory overhead for each application processes is the 4 integers needed for the matrix clock.

The computation overhead consists of monitoring the local variables. The rest of the computation is offloaded to the checker process which uses the following data: $Q0[1 \ldots R_0]$, $Q1[1 \ldots R_1]$, $minx.(0,i)$ for $1 \leq i \leq M_0$, and $minx.(1,j)$ for $1 \leq j \leq M_1$. R_i and M_i are defined in section 5.1.

Consider the call $check(Q0[1 \ldots R_0])$. On a heap of N elements, $insert()$ and $removetop()$ each cost $\Omega(\lg N)$ and $top()$ costs $\Omega(1)$. Each element of $minx.(0,\cdot)$ is inserted at most once and removed at most once from $heap_0$ for a total cost of $\Omega(M_0 \lg M_0)$. Similarly, the total cost of operations on $heap_1$ is $\Omega(M_1 \lg M_1)$. The outer loop executes R_0 times but is added to the cost of the heap operations since the heap operations are spread out through all R_0 iterations. Thus the total cost of $check(Q0[1 \ldots R_0])$ is $\Omega(R_0 + M_0 \lg M_0 + M_1 \lg M_1)$. Since $R_0 \leq M_0$, this simplifies to $\Omega(M \lg M)$ where $M = \max(M_0, M_1)$. Since $check$ is only called twice, the total complexity is $\Omega(M \lg M)$.

7 Generalization

In this section, let σ_i represent a state in S_i. The algorithm given in this paper detects the predicate $(\exists \sigma_0, \sigma_1 : \sigma_0 \parallel \sigma_1 : \sigma_0.x + \sigma_1.x < C)$. Our approach was to compute the minimum values of $\sigma_0.x$ and $\sigma_1.x$ and compare their sum to C. Since $+$ distributes over min (ie $a + min(b,c) = min(a+b, a+c)$), this is equivalent to computing

$$(\min \sigma_0, \sigma_1 : \sigma_0 \parallel \sigma_1 : \sigma_0.x + \sigma_1.x)$$

and comparing it to C. The algorithm presented in this paper repeatedly calculates the above expression for different segments of a run; due to the idempotency of min it could be easily modified to determine the value of the expression for the entire run. The above expression uses min and $+$ over integers. This can be generalized to two operators, $\oplus$ and $\otimes$, over a domain D which meet the following requirements:

Domain	D
Addition	$\oplus : D \times D \mapsto D$
Multiplication	$\otimes : D \times D \mapsto D$
Commutativity	$a \oplus b = b \oplus a$
Associativity	$a \oplus (b \oplus c) = (a \oplus b) \oplus c$
Idempotency	$a \oplus a = a$
Distributivity	$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$

Let $\sigma_i.x$ denote the value of a variable x with domain D in state $\sigma_i \in S_i$. Then our algorithm calculates

$$(\oplus \sigma_0, \sigma_1 : \sigma_0 \parallel \sigma_1 : \sigma_0.x \otimes \sigma_1.x)$$

This generalization is very useful as shown by the following examples.

Example 1 $(D, \oplus, \otimes) := (Integers, \min, +)$.
The resulting calculation is $(\min \sigma_0, \sigma_1 : \sigma_0 \parallel \sigma_1 : \sigma_0.x + \sigma_1.x)$. This is the construction used in the presentation of the algorithm.

■

Example 2 $(D, \oplus, \otimes) := (Reals, \max, *)$.
The resulting calculation is $(\max \sigma_0, \sigma_1 : \sigma_0 \parallel \sigma_1 : \sigma_0.x * \sigma_1.x)$ which is the maximum value of $\sigma_0.x * \sigma_1.x$ in any consistent cut.

■

Example 3 $(D, \oplus, \otimes) := (\{T, F\}, \vee, \wedge)$.
The resulting calculation is $(\vee \sigma_0, \sigma_1 : \sigma_0 \parallel \sigma_1 : \sigma_0.x \wedge \sigma_1.x)$. This is equivalent to weak conjunction [GW94] and "possibly $\sigma_0.x \wedge \sigma_1.x$" [CM91].

■

Example 4 $(D, \oplus, \otimes) := (\{T, F\}, \wedge, \vee)$.
The resulting calculation is $(\wedge \sigma_0, \sigma_1 : \sigma_0 \parallel \sigma_1 : \sigma_0.x \vee \sigma_1.x)$. This is the dual of example 3 and could be called strong disjunction: in every cut either $\sigma_0.x$ or $\sigma_1.x$ is true.

■

Example 5 $(D, \oplus, \otimes) := (\{T, F\}, \wedge, \wedge)$.
The resulting calculation is $(\wedge \sigma_0, \sigma_1 : \sigma_0 \parallel \sigma_1 : \sigma_0.x \wedge \sigma_1.x)$ which states that both $\sigma_0.x$ and $\sigma_1.x$ are invariant.

■

8 Conclusion

Relational global predicates (as defined in this paper) are of the form $(x_0 + x_1 + \ldots + x_n < C)$ where x_i is an integer valued variable in process P_i. We presented decentralized and centralized algorithms to efficiently detect the occurrence of a restricted form of relational global predicates (i.e., $x_0 + x_1 < C$) in a distributed system of N processes.

The algorithms are developed and formally proven with a model presented in the paper. The model makes extensive use of intervals (sequences at a single process in between message activity) and of the successor and predecessor functions (maps local states to the "next" or "previous" local states at another process). We assume unordered asynchronous message channels.

The decentralized algorithm runs concurrently with the target program and can be used for online detection of the predicate. It uses constant size message tags (4 integers) and has low message overhead.

The centralized version is decoupled from from the target program and can be run concurrently with the target program or after it terminates. This version uses the same message tags as the decentralized version and has $O(M \log M)$ complexity, where M is the number of messages received $P0$ and $P1$.

Relational global predicates cannot be efficiently detected online (i.e., while the target program executes) by algorithms presented in earlier work. Earlier work suffers from one or more of the following problems: cannot express relational global predicates, cannot detect them online, or has exponential complexity.

We generalized our results to an algebra $(D, \oplus, \otimes)$ where $\oplus$ and $\otimes$ are binary operators in domain D, $\oplus$ is commutative, associative and idempotent, and $\otimes$ distributes over $\oplus$. In this algebra we can calculate the value of the expression $(\oplus\sigma_0, \sigma_1 : \sigma_0 \parallel \sigma_1 : \sigma_0.x \otimes \sigma_1.x)$. This generalization opens up many useful variants of our algorithm, including detection of weak conjunctive predicates and their duals, which we call strong disjunctive predicates.

9 Acknowledgments

We are grateful to Ken Marzullo, Michel Raynal and Jong-Deok Choi for helpful comments on earlier versions of this paper.

References

[Bou87] L. Bouge. Repeated snapshots in distributed systems with synchronous communication and their implementation in CSP. *Theoretical Computer Science*, 49:145–169, 1987.

[CL85] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

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

[Fid89] C. J. Fidge. Partial orders for parallel debugging. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):183–194, January 1989.

[Gar92] V.K. Garg. Some optimal algorithms for decomposed partially ordered sets. *Information Processing Letters*, 44:39–43, November 1992.

[GW92] V.K. Garg and B. Waldecker. Detection of unstable predicates in distributed programs. In *Proc. of 12th Conference on the Foundations of Software Technology & Theoretical Computer Science*, pages 253–264. Springer Verlag, December 1992. Lecture Notes in Computer Science 652.

[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.

[HPR93] M. Hurfin, N. Plouzeau, and M. Raynal. Detecting atomic sequences of predicates in distributed computations. In *Proc. of the Workshop on Parallel and Distributed Debugging*, pages 32–42, San Diego, CA, May 1993. ACM/ONR. (Reprinted in SIGPLAN Notices, Dec. 1993).

[HW88] D. Haban and W. Weigel. Global events and global breakpoints in distributed systems. In *Proc. of the 21^{st} International Conference on System Sciences*, volume 2, pages 166–175, January 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: Proc. of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V. (North-Holland), 1989.

[Ray92] M. Raynal. About logical clocks for distributed systems. *ACM Operating Systems Review*, 26(1):41–48, 1992.

[SK86] M. Spezialetti and P. Kearns. Efficient distributed snapshots. In *Proc. of the* 6^{th} *International Conference on Distributed Computing Systems*, pages 382–388, 1986.

[SK88] M. Spezialetti and P. Kearns. A general approach to recognizing event occurrences in distributed computations. In *Proc. of the* 9^{th} *International Conference on Distributed Computing Systems*, pages 300–307, 1988.