to CD
1/09/85

# A PARADIGM FOR DETECTING QUIESCENT PROPERTIES IN DISTRIBUTED COMPUTATIONS

K. Mani Chandy

Jayadev Misra

Department of Computer Sciences
University of Texas at Austin
Austin, Texas, 78712

9 January 1985

This work was supported in part by a grant from the Air Force Office of Scientific Research under AFOSR 810205.

# Table of Contents

1. Introduction 1
2. Model of Distributed Systems 1
    2.1. The Model 1
    2.2. Quiescent Property 2
    2.3. Problem Definition 3
    2.4. Termination 3
    2.5. Database Deadlock 3
    2.6. Communication Deadlock 4
    2.7. Systems with Acknowledgements 5
3. The Paradigm 5
    3.1. Quiescence Detection Paradigm 6
    3.2. Proof of Correctness 7
    3.3. Implementation of the Paradigm 7
        3.3.1. Systems with Acknowledgements 7
        3.3.2. Systems with First-In-First-Out Channels 8
    3.4. Notes on the Paradigm 9
4. Applications of the Paradigm 9
    4.1. Termination Detection 9
        4.1.1. The Algorithm 10
        4.1.2. Proof of Correctness 12
        4.1.3. Overhead and Efficiency 13
    4.2. Deadlock Detection 13
5. Previous Work 14

## 1. Introduction

The problem of stability detection is one of the most widely studied problems in distributed computing [ 1-28 ]. A *stable property* is one that persists: if the property holds at any point, then it holds thereafter. Examples of stable properties are termination, deadlock and loss of tokens in a token-ring. The problem is to devise algorithms to be superimposed on the underlying computation to determine whether a specified stable property holds for the underlying computation. This paper presents a simple (almost trivial) algorithm to detect *quiescent properties*, an important class of stable properties including those mentioned above. Distributed snapshots [ 7 ] may be used to derive algorithms for these problems. However our approach in this paper is different and results in simpler algorithms.

## 2. Model of Distributed Systems

### 2.1. The Model

A distributed system is a set of processes and a set of directed communication channels. Each channel is directed from one process to another process. Processes send messages on outgoing channels and receive messages on incoming channels. A process sends a message along an outgoing channel by depositing it in the channel. A process receives a message along an incoming channel by removing the message from the channel. A process may receive a message some arbitrary time after it is sent. Initially, all channels are *empty*. At any time each process is in one of a set of process states and each channel is in one of a set of channel states. The channel state for a first-in-first-out channel is the sequence of messages in transit along the channel. For channels which deliver messages in arbitrary order, the channel state is the set of messages in transit. A system has a set of states, an initial state from this set, and a set of state transitions. The system state at any time is the set of process and channel states. Let $S$, $S^*$ be states of a system. $S^*$ is *reachable* from $S$ if and only if there exists a sequence of state transitions from $S$ to $S^*$. We assume that all system states are reachable from the initial system state.

### 2.2. Quiescent Property

A *stable property* $B$ of a distributed system is a predicate on system states such that for all $S^*$ reachable from $S$:

$B(S)$ *implies* $B(S^*)$

In other words, once a stable property becomes *true* it remains *true*. A *quiescent* property of a distributed system is a special kind of stable property characterized by (1) a subset $P^*$ of the set of processes, (2) for all processes $p$ in $P^*$, a predicate $b_p$ on the process states of $p$ and (3) a subset $C^*$ of the set of channels between processes in $P^*$. A process $p$ in $P^*$ cannot send messages along channels in $C^*$ while $b_p$ holds. Furthermore, if $b_p$ is *true*, it must remain *true* at least until $p$ receives a message along a channel in $C^*$. The quiescent property $B$ is:

all channels in $C^*$ are *empty* and for all processes $p$ in $P^*$: $b_p$.

It is easily seen that $B$ is also a stable property. A process $p$ is a *predecessor* of a process $q$ with respect to $B$ if and only if $p$ and $q$ are both in $P^*$ and there exists a channel in $C^*$ from $p$ to $q$. For brevity we shall say $p$ is a predecessor of $q$ and drop the phrase "with respect to $B$". If for some system state, we have for some process $q$ in $P^*$:

$b_q$ *and* all of $q$'s incoming channels in $C^*$ are *empty and*

for all predecessors $p$ of $q$: $b_p$ (1)

then this condition must persist at least until for some predecessor $p$ of $q$, $b_p$ becomes *false*. This fact is useful in understanding quiescent properties and their detection.

### 2.3. Problem Definition

Let the system computation go through a sequence of global states $S_i$, $i \geq 0$, where $S_0$ is the initial state; this sequence of global states will be called the *underlying computation*. Given a quiescent property $B$ we wish to superimpose a detection algorithm on the underlying computation to determine whether $B$ holds. The detection algorithm sets a boolean variable *claim* to *true* when it detects that $B$ holds, and *claim* is *false* until that point. The detection algorithm must guarantee:

(Safety) : *not claim or* $B$

(Liveness) : within finite time of $B$ becoming *true, claim* is set to *true.*

We now present a brief discussion of three instances of quiescent properties: termination, database deadlock and communication deadlock.

### 2.4. Termination

A computation is defined to be *terminated* if and only if all processes are *idle* and all channels are *empty.* Thus $C^*$ is the set of all channels, $P^*$ is the set of all processes, and for each process $p$, $b_p$ is: $p$ is *idle.* Idle processes don't send messages and hence termination is a quiescent property.

### 2.5. Database Deadlock

A process is either *active* or *waiting.* A waiting state of a process $p$ is specified by a pair $(R_p, H_p)$ where $R_p$ is a non-empty set of resources that $p$ is waiting for and $H_p$ is a set of resources that $p$ needs and holds (where $R_p$ and $H_p$ have no common elements). Resources are sent as messages from active processes to other processes; a waiting process does not send any resource it needs and holds. A process $p$, in a waiting state specified by $(R_p, H_p)$, takes the following action on receiving a resource $r$ in $R_p$:

**begin** $R_p := R_p - \{r\}$; $H_p := H_p \cup \{r\}$;

**if** $R_p = \{\}$ **then** become active **else** wait

**end**

Here { } is the empty set. When $p$ transits from active to waiting state, $R_p$ and $H_p$ are set to values which are of no consequence to us here. A set $P^*$ of processes is deadlocked if every process in $P^*$ is waiting for resources held by other processes in $P^*$, i.e.

$P^*$ is database deadlocked $\equiv$

for all $p$ in $P^*$: $p$ is waiting and there exists a $q$ in $P^*$ such that

$R_p \cap H_q \neq \{\}$

In this case, the predicate $b_p$ is: $p$ is waiting for $R_p$ and $p$ holds $H_p$. A channel $c$ is in $C^*$ if and only if $c$ is from a process $q$ to a process $p$ where $p$ and $q$ are both in $P^*$, and $q$ holds a resource required by $p$. Typically, $P^*$ is not specified and it is required to obtain a $P^*$ as part of the detection algorithm.

### 2.6. Communication Deadlock

As in database deadlock a process is *active* or *waiting*. A waiting process $p$ is waiting on a set of incoming channels $C_p$; on receiving a message along *any* channel in $C_p$, process $p$ becomes active. An active process may start waiting at any time. Until it receives a message along a channel in $C_p$, a waiting process $p$ continues to wait on $C_p$. A waiting process cannot send messages. A set of waiting processes is deadlocked if no process in the set is waiting on a channel from a process outside the set, and all channels between processes in the set are *empty*, i.e.,

A set of processes $P^*$ is communication deadlocked $\equiv$

for all $p$ in $P^*$: $p$ is waiting for a set of incoming channels $C_p$ where each channel $c$ in $C_p$ is from a process in $P^*$, and $c$ is *empty*.

In this case, $b_p$ is: $p$ is waiting on $C_p$. $C^*$ is the union of all $C_p$ for $p$ in $P^*$.

As in database deadlock, the detection algorithm is required to find $P^*$ if such a set exists. Next we consider two specific classes of distributed systems: (i) systems in which messages are acknowledged and (ii) systems in which channels

are first-in-first-out, and show how to detect quiescent properties in each class. The latter class needs little description. We describe the former class next.

### 2.7. Systems with Acknowledgements

Let $c$ be a channel from a process $p$ to a process $q$. On receiving a message along $c$, process $q$ sends an acknowledgement $ack_c$ to $p$. We are not concerned with how *acks* travel from one process to another. An *ack* is not considered to be a message in that *acks* are not acknowledged in turn. Furthermore, the statement "channel $c$ is *empty*" means that $c$ contains no message; it may or may not contain *acks*. Let $num_c$ be the number of unacknowledged messages $p$ has sent along outgoing channel $c$, i.e.,

$num_c$ = number of messages sent by $p$ along $c$ −
number of $ack_c$ acknowledgements received by $p$.

$num_c = 0$ *implies* $c$ is *empty*.

We assume that every message sent is received in finite time and acknowledged in finite time. We also assume that every *ack* sent is received in finite time. Hence, an acknowledgement is received for each message within finite time of sending the message. Therefore,

if $B$ becomes *true*, then within finite time of $B$ becoming *true*:

for all $c$ in $C^*$: $num_c = 0$

## 3. The Paradigm

Our paradigm is based on observing each process computation for some period of time called an observation period. An *observation period* for a process $p$ is specified by two integers, $start_p$ and $end_p$, $start_p \leq end_p$, denoting that $p$'s computation is observed at every $S_i$, $start_p \leq i \leq end_p$. An *observation period set* for a quiescent property $B$ is a set of observation periods, one for each process in $P^*$.

An observation period set $obs'' = \{(start_p'', end_p'') \mid p \text{ in } P^*\}$ *is later than* an

observation period set $obs' = \{(start_p', end_p') \mid p \text{ in } P^*\}$ if and only if all starting times in $obs''$ are after some starting time in $obs'$, i.e.

$$\min_p \; start_p'' \quad > \quad \min_p \; start_p'$$

Let $B^*$ be a predicate on observation period sets, defined as follows.

$B^*(obs) \equiv$ [for all $p$ in $P^*$ :

for all states $S_i$ where $start_p \leq i \leq end_p$ : $b_p$ holds in $S_i$]

*and*

[for all $p,q$ in $P^*$ where $p$ is a predecessor of $q$: all messages sent

by $p$ at or before $start_p$ are received by $q$ at or before $end_q$] (2)

**Note:** To ensure that messages sent by $p$ at or before $start_p$ are received by $q$ at or before $end_q$, we must have for all $p,q$ in $P^*$ where $p$ is a predecessor of $p$ : $start_p \leq end_q$ (3)

### 3.1. Quiescence Detection Paradigm

*claim* : = *false*; obtain an observation period set *obs*;

**while** *not* $B^*(obs)$ **do**

obtain an observation period set *obs'* later than *obs*;

*obs* : = *obs'*

**od**;

*claim* : = *true*

We next prove the correctness of this paradigm and postpone discussion of techniques for implementing the paradigm to a later section.

### 3.2. Proof of Correctness

Safety : *not claim or B*

Safety holds while *claim* is *false*; therefore consider the final iteration of the while loop after which *claim* is set to *true*. For this iteration, we prove the following by inducting on $i$:

for all $i \geq 0$ : for all $p$ in $P^*$ :

[ [ $i < start_p$ or $b_p$ holds in $S_i$ ] *and*

[ $i < end_p$ or $p$'s incoming channels in $c$ are empty ] ]

This induction follows from (1), (2), and (3).

Liveness: If there exists an $i \geq 0$ such that $B$ holds for $S_i$ then there exists a $j \geq 0$ such that $claim = true$ in $S_j$ . If $B$ holds for $S_i$ then for all observation period sets, *obs*, where $start_p \geq i$, for all $p$, $B^*(obs)$ holds. From the paradigm, either *claim* is set *true* or later observation periods are chosen indefinitely. Hence if $B$ holds for $S_i$ for any $i \geq 0$, then *claim* will be *true* for some $S_j$, $j \geq 0$.

### 3.3. Implementation of the Paradigm

The key question for implementation is: How can we ensure that all messages sent by a predecessor $p$ of a process $q$ at or before $start_p$, are received by $q$ at or before $end_q$?

#### 3.3.1. Systems with Acknowledgements

The above question can be answered for systems with acknowledgements by ensuring the following condition: for all $p,q$ in $P^*$ where $p$ is a predecessor of $q$ and for all channels $c$ from $p$ to $q$ :

$num_c$=0 at $start_p$ *and* $start_p \leq end_q$.

Proof of this condition is as follows. At $start_p$, $num_c$=0 implies that $c$ is empty and hence all messages sent along $c$ have been received. Hence all messages sent at or before $start_p$, along $c$, are received at or before $start_p$ and

since $start_p \leq end_q$, the result follows.

For all $p$ in $P^*$, let $quiet_p \equiv$ for all states $S_i$, where $start_p \leq i \leq end_p$ : [$b_p$ and for all outgoing channels $c$ in $C^*$ : $num_c = 0$].

In the paradigm we replace $B^*(obs)$ by

[for all $p$ in $P^*$ : $quiet_p$] *and*

[for all $p,q$ in $P^*$ where $p$ is a predecessor of $q$ : $start_p \leq end_q$].

We show, in section 4, how $start_p \leq end_q$, can be maintained. $num_c$ is maintained as a local variable of $p$ and hence $quiet_p$ can be determined by $p$. Note that for systems with rendezvous, such as CSP and ADA, $num_c$=0 holds at all times.

### 3.3.2. Systems with First-In-First-Out Channels

To answer the key question posed at the beginning of this section, we use special messages called *markers*, which are sent and received along channels in $C^*$. They have no effect on the underlying computation other than that they occupy the same channels as regular messages. We use the following implementation rules.

**R1.** Every process $p$ in $P^*$ sends one marker along each outgoing channel in $C^*$ some (finite) time after (or at) $start_p$ and,

**R2.** Every process $p$ in $P^*$ has received one marker along each incoming channel in $C^*$ some time before (or at) $end_p$.

Since channels are first-in-first-out, all messages sent along a channel before the marker is sent on the channel must be received before the marker is received. Hence every message sent at or before $start_p$ is received at or before $end_q$, for all $p,q$ in $P^*$, where $p$ is a predecessor of $q$.

Each process $p$ in $P^*$ maintains a local boolean variable $quiet_p$ where

$quiet_p \equiv$ for all states $S_i$ where $start_p \leq i \leq end_p : b_p$.

In the paradigm we replace $B^*(obs)$ by : [for all $p$ in $P^*$ : $quiet_p$] and rules **R1**, **R2** are satisfied.

### 3.4. Notes on the Paradigm

Our constraints on observation period sets are weak. For instance it is possible that for a predecessor $p$ of $q$, $start_q > end_p$ and there may be *no overlap* between $p$'s and $q$'s observation periods. For a system with first-in-first-out channels, process $p$ may send markers on some or all outgoing channels *after* $end_p$, and may receive markers on some or all incoming channels *before* $start_p$.

If the quiescent property never holds, the iteration in the paradigm will never terminate, i.e. an infinite sequence of observation period sets will be obtained.

## 4. Applications of the Paradigm

There are many problems to which the paradigm may be applied and many ways of applying the paradigm. We show two examples to demonstrate the power of the paradigm: termination detection and (both types of) deadlock detection, described earlier. We use termination detection as an example of the use of markers and deadlock detection as an example of the use of acks.

### 4.1. Termination Detection

Processes are labeled $p_i$, $0 \leq i < n$. We employ a *token* to transmit the values $quiet_p$. The token cycles through the processes visiting $p_{(i+1) mod\ n}$ after departing from $p_i$, all $i$. A cycle is initiated by a process $p_{init}$, called the initiator. If the token completes a cycle (i.e. returns to $p_{init}$ after visiting all processes) and if all processes $p$ return a value $quiet_p$ of *true* in this cycle then the initiator detects termination, i.e. it sets *claim* to *true*. If any process $q$ returns a value $quiet_q$ of *false* in a cycle, then the current cycle is terminated and

a new cycle is initiated with $q$ as the initiator. A process ends one observation period and immediately starts the next observation period when it sends the token. The algorithm, described next in detail, shows how $quiet_p$ is set.

### 4.1.1. The Algorithm

The are no shared variables in a distributed system. However, *for purposes of exposition* we assume that *claim* is a shared global variable which has an initial value of *false* and which may be set *true* by any process. Such a global variable can be simulated by message transmissions; for instance, the process that sets *claim* to *true* may send messages to all other processes notifying them.

Two types of messages are employed in the termination detection algorithm.

<marker> : this type of message has already been discussed; it carries no other information (except its own type).

<token, initiator> : this is the token and its initiator, as described in Section 4·1.

Each process has the following constants and variables. These will be subscripted, by $i$, when referring to a specific process $i$.

*ic*: number of incoming channels to the process, a constant,

*idle*: process is idle,

*quiet*: process has been continuously idle since the token was last sent by the process; *false* if the token has never been sent by this process,

*hold-token*: process holds the token,

*init*: the value of initiator in the <token, initiator> message last sent or received; undefined if the process has never received such a message,

*m*: number of markers received, since the token was last sent by the process; initial value as given in the algorithm.

**Initial Conditions**

The token is at $p_0$.

$m_i$ = the number of channels from processes with indices greater than $i$, for all $i$, i.e., the cardinality of the set, $\{c \mid c$ is a channel from $p_j$ to $p_i$ *and* $j > i\}$.

(This initial condition is required because otherwise, the token will permanently stay at one process.)

$quiet_i$ = *false*, for all $i$.

(The algorithm is slightly more efficient with different initial conditions, but for purposes of exposition we shall make the simpler assumption.)

$$hold\text{-}token_i = \begin{cases} true, \text{ for } i=0 \\ false, \text{ for } i \neq 0 \end{cases}$$

$init_i$ is arbitrary, for all $i$

**Algorithm for a Process $P_i$**

The algorithm for a process is a repetitive guarded command. The repetitive guarded command is a set of rules where each rule is of the form, *condition* → *action*. The algorithm proceeds as follows: one of the rules whose condition part evaluates to *true* is selected nondeterministically and its action part is executed. The repetitive guarded command consists of the following rules:

1. receive *marker* → $m_i := m_i + 1$;

2. $quiet_i$ *and* receive regular message (i.e. underlying computation's message) → $quiet_i := false$;

3. receive <*token, initiator*> → **begin** $init_i := initiator$; $hold\text{-}token_i := true$ **end**;

4. $hold\text{-}token_i$ *and* $(ic_i = m_i)$ *and* $idle_i \rightarrow$

**if** $quiet_i$ *and* $(init_i = i)$ **then** {termination detected} $claim := true$;

**if** $quiet_i$ *and* $(init_i \neq i)$ **then** {continue old cycle}

**begin**

$m_i := 0$ ;

Send marker along each outgoing channel;

$hold\text{-}token_i := false$;

*send* $<token, init_i>$ to $p_{(i+1) mod\ n}$

**end**

**if** $\sim quiet_i$ **then** {initiate new cycle}

**begin**

$m_i := 0$; $quiet_i := true$; $init_i := i$;

Send marker along each outgoing channel;

$hold\text{-}token_i := false$;

send $<token, init_i>$ to $p_{(i+1) mod\ n}$

**end**

### 4.1.2. Proof of Correctness

We need merely show that the algorithm fits the paradigm. A process $p_i$ ends an observation period and starts the next one when the token leaves $p_i$. Initially, an observation period is started when the token leaves $p_0$; the values of $m_i$ are so chosen initially that it is possible for the token to leave $p_i$, for the first time, when $p_i$ has received markers from all lower numbered processes. We need to show the following {initial conditions should be treated slightly differently}:

1. $quiet_i \equiv$ $p_i$ has been continuously idle in the current observation period, i.e. since the token last left $p_i$

2. Each process sends a marker on each outgoing channel upon starting an observation period.

3. Each process ends an observation period only after receiving exactly one marker along each incoming channel.

4. *claim* is set to *true* if and only if in one cycle of the token (which corresponds to an iteration of the paradigm all processes $p_i$ return a value of $quiet_i = true$ at the end of their observation periods.

5. After termination, a cycle of the token is completed in finite time. To guarantee this we must ensure that each process receives a marker along each incoming channel in finite time.

Proofs of these assertions follow directly from the algorithm and the details are left to the reader.

### 4.1.3. Overhead and Efficiency

The most overhead is incurred in rule 4, when a process is idle. The overhead while a process is doing useful work in negligible. Also a process sends the token only when the process is idle; this controls the rate at which the token cycles through processes. For instance, if all processes are active, the token will not move at all. Also observe that termination will be detected within two cycles of after computation terminates.

## 4.2. Deadlock Detection

The following refinement of the paradigm is applicable to database deadlock and communication deadlock, under the assumption that messages are acknowledged.

A process which we call the *detector* sends *initiate* messages to all processes; on receiving an *initiate* message a process starts its observation period and acknowledges the *initiate* message. After receiving acknowledgements to all the *initiate* messages sent the detector sends *finish* messages to all processes. A process $p$ ends its observation period after receiving a *finish* message and replies with a boolean value $quiet_p$ and a set $waiting\text{-}for_p$, where

$quiet_p \equiv$ for all states in the observation period :
[$p$ is *waiting and* for all outgoing channels $c$ : $num_c = 0$]

$$waiting\text{-}for_p = \begin{cases} \text{set of objects that } p \text{ is waiting for in the observation period, if } quiet_p. \\ \\ \text{arbitrary, if } not\ quiet_p \end{cases}$$

The *detector* determines whether there exists a set of processes $P^*$, such that for all $p$ in $P^*$ : $quiet_p$ and the sets $waiting\text{-}for_p$ are such as to constitute a deadlock. The proof of correctness is that the algorithm fits the paradigm.

The algorithm, as stated above, appears to be centralized rather than distributed. Note however, that the *detector* process could be different for different initiations and there could be multiple *detectors*. The function of the detector, i.e. sending messages, detecting deadlock, can be decentralized by having messages forwarded to their destinations by intermediate processes and deadlock detection computation carried out by intermediate processes.

## 5. Previous Work

The idea of observation periods is central to the works of Francez, Rodeh and Sintzoff on distributed termination [ 12-14 ], and Chandy, Misra and Haas on deadlock detection [ 6 ]. Dijkstra [ 11 ], Gouda [ 16 ] and Misra [ 26 ] have developed token based algorithms for termination detection, and these algorithms also use observations over a period. We have attempted to generalize these works to produce a particularly simple paradigm for detecting an important class of properties, quiescent properties, in distributed systems with asynchronous channels.

**Acknowledgement:** It is a pleasure to acknowledge extensive discussions with Shmuel Katz, who also carefully read and commented on earlier draft of this paper. Edsger W. Dijkstra and Hank Korth helped with constructive criticism.

## References


1. C. Beeri and R. Obermarck, "A Resource Class Independent Deadlock Detection Algorithm", *Research Report RJ3077,* IBM Research Laboratory, San Jose, California, May 1981.

2. G. Bracha and S. Toueg, "A Distributed Algorithm For Generalized Deadlock Detection", *Technical Report TR 83-558,* Cornell University, June 1983.

3. K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation Via a Sequence of Parallel Computations", *Communications of the ACM,* Vol. 24, No. 4, pp. 198-205, April 1981.

4. K. M. Chandy and J. Misra, "A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems", *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing,* Ottawa, Canada, August 1982.

5. K. M. Chandy and J. Misra, "A Computation on Graphs: Shortest Path Algorithms", *Communications of the ACM,* Vol. 25, No. 11, pp. 833-837, November 1982.

6. K. M. Chandy and J. Misra and L. Haas, "Distributed Deadlock Detection", *ACM Transactions on Computing Systems,* Vol. 1, No. 2, pp. 144-156, May 1983.

7. K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems", to appear in *ACM Transactions on Computing Systems.*

8. E. Chang, "Echo Algorithms: Depth Parallel Operations on General Graphs", *IEEE Transactions on Software Engineering,* Vol. SE-8, No. 4, pp. 391-401, July 1982.

9. S. Cohen and D. Lehmann, "Dynamic Systems and Their Distributed Termination", *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing,* pp. 29-33, Ottawa, Canada, August 18-20, 1982.

10. E. W. Dijkstra and C. S. Scholten, "Termination Detection for Diffusing Computations", *Information Processing Letters,* Vol. 11, No. 1, August 1980.

11. E. W. Dijkstra, "Distributed Termination Detection Revisited", EWD 828, Plataanstraat 5, 5671 AL Nuenen, The Netherlands.

12. N. Francez, "Distributed Termination", *ACM Transactions on Programming Languages and Systems,* Vol. 2, No. 1, pp. 42-55, January 1980.

13. N. Francez, M. Rodeh, and M. Sintzoff, "Distributed Termination with Interval Assertions", *Proceedings of Formalization of Programming Concepts,* Peninusla, Spain, April 1981. Lecture Notes in Computer Science 107, (Springer-Verlag).

14. N. Francez and M. Rodeh, "Achieving Distributed Termination Without Freezing", *IEEE-TSE,* Vol. SE-8, No. 3, pp. 287-292, May 1982.

15. V. Gligor and S. Shattuck, "On Deadlock Detection in Distributed Data Bases", *IEEE Transactions on Software Engineering,* Vol. SE-6, No. 5, September 1980.

16. M. Gouda, "Personal Communication", Department of Computer Sciences, University of Texas, Austin, Texas 78712.

17. L. Haas and C. Mohan, "A Distributed Deadlock Detection Algorithm for a Resource-Based System", *Research Report RJ3765,* IBM Research Laboratory, San Jose, California, January 1983.

18. T. Herman and K. M. Chandy, "A Distributed Procedure to Detect AND/OR Deadlock", Computer Sciences Department, University of Texas, Austin, Texas 78712, February 1983.

19. T. Holt, "Some Deadlock Properties of Computer Systems", *Computing Surveys,* Vol. 4, No. 3, pp. 179-196, September 1972.

20. D. Kumar, Ph.D Theses (in preparation), Computer Sciences Department, University of Texas, Austin, Texas 78712.

21. L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System", *Communications of the ACM,* Vol. 21, No. 7, July 1978.

22. G. Le Lann, "Distributed Systems - Towards a Formal Approach", *Information Processing 77,* IFIP, North-Holland Publishing Company, 1977.

23. D. Menasce and R. Muntz, "Locking and Deadlock Detection in Distributed Data Bases", *IEEE Transactions on Software Engineering,* Vol. SE-5, No. 3, May 1979.

24. J. Misra and K. M. Chandy, "A Distributed Graph Algorithm: Knot Detection", *ACM Transactions on Programming Languages and Systems,* Vol. 4, No. 4, pp. 678-688, October 1982.

25. J. Misra and K. M. Chandy, "Termination Detection of Diffusing Computations in Communicating Sequential Processes", *ACM Transactions on Programming Languages and Systems,* Vol. 4, No. 1, pp. 37-43, January 1982.

26. J. Misra, "Detecting Termination of Distributed Computations Using Markers", *Proceedings of the ACM SIGACT-SIGOPS Symposium of Principles of Distributed Computing,* Montreal, Canada, August 17 - 19, 1983.

27. R. Obermarck, "Deadlock Detection For All Resource Classes", *Research Report RJ2955,* IBM Research Laboratory, San Jose, California, October 1980.

28. R. Obermarck, "Distributed Deadlock Detection Algorithm", *ACM Transactions on Database Systems,* Vol. 7, No. 2, pp. 187-208, June 1982.