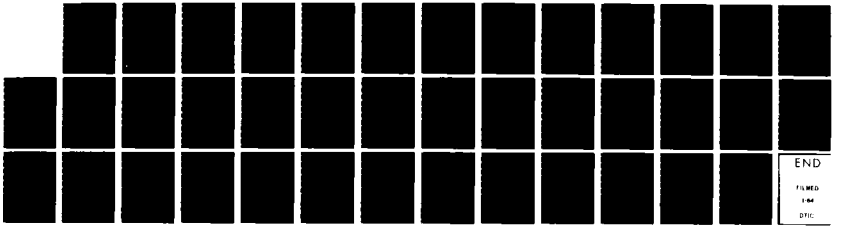
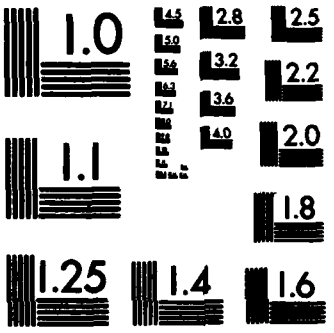
AD-A135 459 A DISTRIBUTED PROCEDURE TO DETECT AND/OR DEADLOCK(U) 1/1
TEXAS UNIV AT AUSTIN DEPT OF COMPUTER SCIENCES
T HERMAN ET AL. JAN 83 AFOSR-TR-83-0989 AFOSR-81-0205
UNCLASSIFIED F/G 9/2 NL





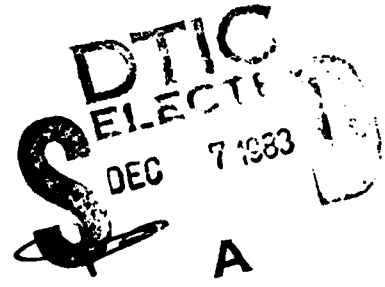
AFOSR-TR- 83-0989

(3)

A DISTRIBUTED PROCEDURE TO DETECT AND/OR DEADLOCK

T. Herman and K. M. Chandy
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712
(512)-471-4353

January, 1983



This research was supported in part by a grant from the Air Force Office of Scientific Research under grant ~~AFOSR-810205~~. AFOSR-81-0205

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFOSR-TR- 83-0989	2. GOVT ACCESSION NO.: A135459
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE (and Subtitle): A Distributed Procedure to Detect AND/OR Deadlock	5. TYPE OF REPORT & PERIOD COVERED: interim
	6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Ted Herman and K. Mani Chandy	8. CONTRACT OR GRANT NUMBER(s): AFOSR 81-0205
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Computer Sciences Department, The University of Texas at Austin, Austin, Texas 78712	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F, 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS: AFOSR/NM, Bolling AFB, DC 20332	12. REPORT DATE: JAN 83
	13. NUMBER OF PAGES: pages
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We present a procedure to detect deadlock in a distributed system. The procedure is dynamic and distributed. Deadlock will be correctly detected for general resource requests of the form: "Lock file A and file B at NY or lock file A and file B at LA." The contribution of this paper is that it presents a distributed solution to the deadlock detection problem when requests have AND/OR form.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

A DISTRIBUTED PROCEDURE TO DETECT AND/OR DEADLOCK

ABSTRACT

The authors ~~We~~ present a procedure to detect deadlock in a distributed system. The procedure is dynamic and distributed. Deadlock will be correctly detected for general resource requests of the form: "Lock file A and file B at NY or lock file A and file B at LA". The contribution of this paper is that it presents a distributed solution to the deadlock detection problem when requests have AND/OR form.

Categories and Subject Descriptors:

C.2.4 [Computer-Communication Networks]: Distributed Systems -- Distributed Databases; D.4.1 [Operating Systems]: Process Management -- Deadlocks;

General Terms: algorithms, graphs, distributed data bases, deadlock detection.

1.0 INTRODUCTION

Deadlock detection is an important and interesting problem. It is important to detect and break up deadlock situations in the interest of system throughput. The detection problem is interesting in a graph-theoretic sense, and also because distributed and parallel processing techniques are applicable.

A deadlock situation is the possible result of competition for resources, for example data base transactions requesting exclusive access to files. Deadlocks can be prevented when resource requests are always granted with system-wide atomicity, but in a distributed system such a guarantee is not practical: There are time delays in the interaction of sites of a distributed system.

1.1 Organization

This paper is divided into six parts. Part 1 is a review of prevalent terminology and deadlock detection algorithms. Part 2 defines the model used for this paper. Part 3 presents the deadlock detection procedure. Part 4 discusses some implementation issues, and Part 5 is the concluding section. Part 6 is an appendix devoted to the correctness proof of the detection procedure.

The remainder of Part 1 defines and classifies the subject of deadlock detection. Section 1.2 is a review of the terminology for deadlock detection in the distributed data base environment. Section 1.3 contains a classification of deadlock models. Section 1.4 completes the introduction with a review of some related solutions to deadlock detection.

1.2 Transaction Model

The paradigm used in this paper for discussion of deadlock detection is due to Menasce and Muntz

[2]. Briefly, data base transactions present resource requests to controllers. A resource request may be a request to lock files or may have more abstract meanings. A transaction is blocked from the time it presents a request to a controller until the controller grants the request, and the transaction becomes active. A resource request can be local, or refer to a resource at another site, in which case the transaction is distributed. A distributed transaction is implemented by transaction agents, each of which is the local agent for a given transaction at one site. Inter-site resource requests are always between two agents of the same transaction.

A transaction wait-for graph (TWFG) is a model of resource requests. The vertices of the graph are associated with transaction agents. Directed edges in the graph represent "wait-for" relationships between transactions agents. A vertex with outgoing edge(s) is a blocked transaction agent. A cycle in the graph indicates deadlock -- if a cycle is defined carefully, as we do in Section 2.4.

1.3 Request Models

Figure 1 is a TWFG graph representing a deadlock situation. This example has four transactions, T1-T4, implemented by seven transaction agents. The directed edge from vertex $T1_1$ to vertex $T2_1$ shows that agent $T1_1$ is blocked. Vertex $T2_2$ has no outgoing edges, and therefore is an active (non-blocked) transaction. Vertex $T1_1$ has two outgoing edges, and this indicates that transaction T1 has two outstanding resource requests, and both must be satisfied before transaction T1 becomes active. This type of resource request is called an AND-request, because transaction T1 must have resource 1 and resource 2.

TRH83a-2

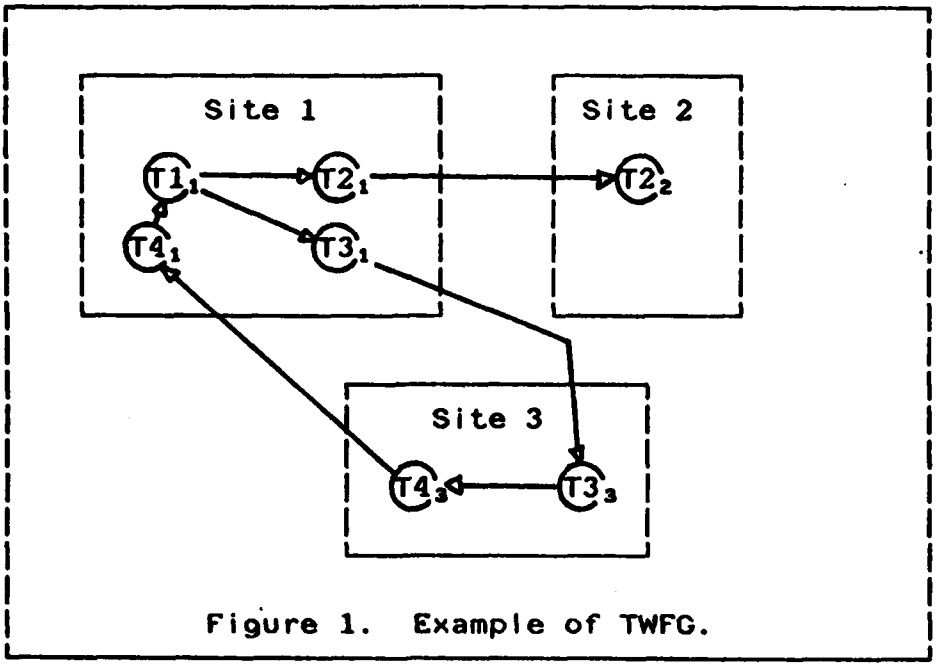


Figure 1. Example of TWFG.

The AND-model is a term used to signify that all resource requests are AND-requests. The AND-model has been the traditional view of resource requests in distributed data base systems.

An alternative model of resource requests is the OR-model. A request for numerous resources is satisfied by the granting of any requested resource. For example, assume that a transaction waits for resource 1 or resource 2. Suppose resource 1 is granted to this request; then the transaction becomes active, and the request for resource 2 is cancelled.

In the OR-model, a cycle is insufficient for deadlock detection. To see this, suppose all requests in Figure 1 are OR-requests; then transaction T1 is not deadlocked because $T2_2$ has no outgoing edges. In terms of the TWFG, a knot will indicate deadlock. A knot is a structure defined as follows: Vertex X is in a knot iff a path from vertex X to vertex Y implies that there is also a

path from Y to X. In other words, in a knot there are no "dead-ends" in the graph.

The AND/OR model, also called the communication model, is a generalization of the two previous models. An AND/OR-request may specify any combination of "and" and "or" in the resource request. For example, a request for "(S and (T or U)) or V" is possible, and S, T, U and V may exist at different sites in the system.

1.4 Related Work

The solution to deadlock detection in a non-distributed system is well studied [10]. It is possible to directly implement this solution in a distributed system if a central scheduler is used, but a central scheduler is not practical. The centralized strategy could theoretically be implemented by broadcasting resource events (requests, grants) with time-stamps as in Lamport [9]. All controllers would have updated global views and deadlock could be detected.

A variation of the time-stamp strategy is proposed in [1]: One controller is designated as the deadlock detector, and all other controllers send pieces of the TWFG, which are time-stamped, to the designated central controller. A consistent, global TWFG is assembled to detect deadlock. The drawbacks associated with a central contoller are ameliorated by structuring the controllers into a hierarchy, which reduces the message traffic, as in [2]. To some extent the hierarchy-of-controllers achieves distribution of processing, although the centralized strategy remains.

A completely distributed algorithm to detect deadlock by construction of a TWFG is non-trivial, as numerous counter-examples show [1,3,5]. The counter-examples typically show that, due to time delays in the transfer of information between controllers, an incorrect TWFG may be constructed.

In this way, deadlock may go undetected or be falsely detected.

The deadlock detection procedure in this paper does not follow the strategy of assembling and manipulating graph structures. Instead, the underlying graph structure is incorporated in the way the program is distributed. Two previous papers employed this theme: A deadlock detection procedure specific to the AND-model appears in [6]. Unlike earlier solutions, the procedure does not construct a TWFG, even in reduced form. In a related work [7], an AND/OR model deadlock detection procedure is developed. This AND/OR deadlock detection procedure is distributed, message-based, and does not construct a TWFG. The procedure efficiently detects deadlock in the OR-model, but is less efficient in an AND/OR-model: Deadlock in the AND/OR-model is detected by repeated application of a test for OR-model deadlock, which will eventually result in detection of deadlock for the AND/OR-model.

A non-distributed solution to the AND/OR model, which contains several interesting examples of AND/OR resource requests, is given by Beeri and Obermarck [4]. The expressive power of the AND/OR model is greater because "non-specific" requests are permitted. For instance, a request for any M available resources, from a pool of size N, can be represented by an AND/OR request. In a multiple-copy distributed data base, a transaction may request the locking of any available copy such as the example in the abstract of this paper suggests. Our contribution is to present a distributed solution to the problem defined by Beeri and Obermarck.

2.0 PROCESS MODEL

This section contains definitions that enhance the basic model presented in Section 1.2. The revised model definition is used to accurately define deadlock and support the discussion of the procedure in Part 3. Sections 2.1 and 2.2 define processes and messages, which are the basic language for the deadlock detection procedure. Section 2.3 describes the wait-for graph for processes, and assigns color attributes to the edges. In Section 2.4 is a definition of deadlock in terms of colored edges. Section 2.5 details the relationship between transactions and processes and Section 2.6 is a list of behavioral axioms for processes.

2.1 Processes

In this paper, the deadlock detection procedure does not directly refer to transactions or agents, but instead to processes. We use the term "process" to extend the idea of transaction agents. Like transaction agents, processes represent transactions. A process can be thought of as a logical entity, manipulated by the controller, much as an operating system manipulates internal tables in the service of jobs. The reason for the distinction is that the mapping from processes to transaction agents is many-to-one, not one-to-one, i.e., the mapping may require that a transaction be represented by numerous processes at a single site, as we will illustrate in Section 2.5.

To simplify the following discussion, the language is "a process does some action" or "a process waits". In fact, it should be understood that the controller is performing actions on behalf of transactions, or to detect deadlock. "A process sends a message to another process" means that a controller sends a message to another controller, only if the processes are at different sites. Sending messages between processes at a single site is an internal operation for a controller.

2.2 Messages

Resource requests and the grants of resources are modeled by messages. To request a resource, a process sends a request (message) to the process that holds the resource. After sending the request the process waits until a grant (message) is received from the relinquishing process. An important assumption about messages is the order of arrival. We assume that all messages sent from one process to another will definitely arrive, and in the order sent (FIFO, loss-free channels between processes). No assumption about the rate of message traffic is made. Other messages will be defined in Part 3 as part of the detection procedure.

2.3 Process Wait-For Graph

Although a wait-for graph is not constructed by the deadlock detection procedure, it will simplify the discussion to refer to this framework. The Process Wait-For Graph (PWFG) is a theoretical construct which reflects the true, instantaneous state of the distributed system. Operations on the PWFG are always theoretical operations. Each vertex in the PWFG represents a process. Assume that each process has a globally unique name, called the process-ID, which is used as the vertex identifier in the PWFG. A directed edge in the PWFG indicates that a process is waiting for a grant message. Edges have colors, attributes that define the state of an edge.

Given processes v and w, edge (v,w) is:

gray — Edge (v,w) is gray from the time v sends w a request message until w receives the request.

black — Edge (v,w) is black from the time w has received a v's request until w sends a grant to v.

white — Edge (v,w) is white from the time w sends a grant to v until v receives the grant.

This notation is fully developed in [6]. To summarize some properties of edges,

- Edges are gray when created.
- A gray edge will turn black after a finite, arbitrary time.
- A white edge will disappear after a finite, arbitrary time.

The normal course of coloring for an edge, over time, is the sequence: non-existence → gray → black → white → non-existence. Attaching colors to edges permits a precise definition of deadlock, as the following section shows.

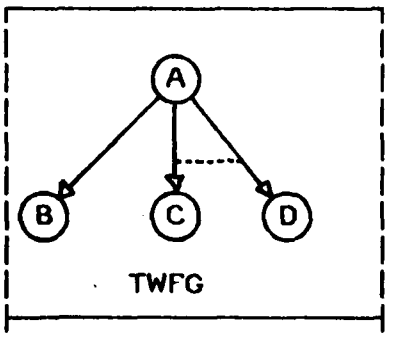
2.4 Deadlock Situation

In the AND-model, a deadlock is identified as a cycle of black edges in the PWFG. More precisely, any combination of gray and black edges in a cycle is deadlock because gray edges turn black [6]. In the OR-model, a deadlock is a black knot. There does not appear to be a familiar term to categorize deadlock in the AND/OR-model, but the idea is similar: A static, black, component arises in the PWFG, which will persist until broken externally.

2.5 Mapping: Transaction → Process

To simplify the presentation in this paper, the PWFG is restricted as follows: A process may have an OR-request or an AND-request. This limitation does not restrict the power of the AND/OR model for the following reason. Although a transaction may submit any type of AND/OR-request, the AND/OR-request can be represented as a network of processes restricted to AND-requests and OR-requests. The mapping is a representation of the AND/OR-request in a regular form, such as disjunctive normal form. Figure 3 shows a TWFG and

the corresponding PWFG as an example of this mapping.



Note: The dashed line between edges is used to indicate an AND-request. In the TWFG, transaction A waits on B or (C and D).

Figure 3. TWFG and PWFG Mapping.

2.6 Process Behavior

This section illustrates the effect of process behavior on the PWFG. An active process is a vertex with no outgoing edges in the PWFG. A blocked process is a vertex with outgoing edges. A blocked process may send neither grant nor request messages. No process, active or blocked, may

receive more than one message at any instant. When a blocked process receives a grant message, an edge in the PWFG disappears and there are several possibilities for the blocked process:

- No outgoing edges remain and the process is active by definition.
- Some outgoing edge(s) remain. There are two cases:
  - The blocked process has an AND-request. The process remains blocked.
  - The blocked process has an OR-request. All of the remaining outgoing edge(s) are deleted from the PWFG, and the process is active.

All of the vanishing edge scenarios listed above are considered to be instantaneous state transitions for the PWFG. That is, all of an OR-request's edges disappear simultaneously in the PWFG (cancellation).

3.0 DEADLOCK DETECTION PROCEDURE

A basic description of the deadlock detection procedure follows. The correctness proof is found in Part 6, and some implementation considerations are in Part 4. Section 3.1 presents notation used to describe the procedure. An informal description of the detection procedure appears in Section 3.2. Section 3.3 is divided into sub-sections that specify the procedure by process behavior rules. In Section 3.4 is an example sequence of an execution of the detection procedure.

3.1 Preliminaries

The deadlock detection procedure is a distributed program. The components of the program are:

- Messages: grants, requests, queries and replies. Grants and requests are discussed in Section 2.2. Queries and replies are used only by the detection procedure and are discussed below.
- Local Memory: Associated with each process are two lists, the outgoing query list (OQ-list) and the incoming query list (IQ-list). These lists will hold images of queries during the deadlock detection procedure. Initially, both lists are empty.

Each query and reply message has a variable length label. The label is a string of process-IDs. Suppose {s,t,u,v,w} are process IDs. A possible label is <twsu>. Some operations on labels are defined:

Catenation: Given two labels, A and B, then A•B denotes the catenation of the strings. Example: A = <uvt>, B = <sw>, A•B = <uvt>•<sw> = <uvtsw>.

Prefix: Given two labels, A and C, then A is prefix or equal to C, A $\preceq$ C, iff A = C

or C can be written as C = A•B, for some label B. Example: A = <uvt>, C = <uvtsw> ⇒ A ⊴ C.

The notation for a query is Q(B,s), where B is the label of query Q which was sent by process s. Query messages are sent in the direction of edges in the PWFG. R(C,t) denotes reply R with label C, sent by process t. Reply messages are sent in the reverse direction of edges in the PWFG.

3.2 Procedure Overview

A controller initiates the procedure, and some period of detection activity follows (sending messages, updating lists and so on). When and how often the procedure is initiated is the topic of Section 4.1. When the activity subsides, the procedure will either have detected deadlock or not.

Queries are used by the procedure to search the PWFG for non-blocked processes. Replies are returned to the senders of queries to say "search failed." When all searches fail, then all replies are returned and deadlock will be detected.

If grant messages or non-blocked processes are encountered in the execution of the detection procedure, then the system is not deadlocked, and therefore the procedure will not detect deadlock.

When a query arrives at a process, then some search is to be extended: The IQ and OQ-lists are updated, and a new query is sent on outgoing edges. The IQ-list is a record of the queries received by a process. The OQ-list is a record of queries sent by a process. Both lists are lists of "on-going" searches: A global, instantaneous view of all IQ and OQ-lists in the distributed system would describe the state of the detection procedure at any instant.

The propagation of queries continues until a cycle is detected. The cycle will be detected because a process will match an arriving query with some

entry in its IQ-list. In a sense, that query has been "seen before" at that process. At such times, reply messages are generated and returned to the query senders.

The queries can be distinguished by their labels for matching purposes, i.e., to generate the "seen before" condition. The query propagation step creates new queries with new labels so that each label carries a "propagation history." Eventually, this history will become exhaustive, giving rise to "seen before" for any process.

When a process receives a reply, then the IQ and OQ-lists are updated (entries removed). The reply signals a "search failed" for activity in the PWFG, and the process propagates the reply to other processes. Each reply is given a label to match a corresponding query label, that is, the "seach failed" message specifies which search failed, and which entry of the OQ-list must be deleted.

3.3 Process Behavior

In the following sections, there is frequently a bifurcation of process actions, depending on whether a process has an OR-request or an AND-request. For convenience, the action for an OR-requesting process will be prefaced with {OR}, and for an AND-requesting process, {AND}.

Only idle (blocked) processes take part in the deadlock detection procedure. Active (non-blocked) processes may simply ignore all request and reply messages, which is implicit in the statement of the following rules.

3.3.1 Initiation of the Procedure

When a controller suspects a deadlock situation, then an artificial process, called the initiator, is created to detect deadlock. Assume that each time a controller creates a new initiator, it is with a different process-ID from all previous ini-

tiator or other process-IDs. The initiator has no edges in the PWFG.

Suppose process i is the initiator, and process w is suspected of being deadlocked. Initiation consists of process i sending Q(<i>,i) to process w. Process i will take no further action during deadlock detection. Later, when R(<i>,w) is received by process i, then the controller declares deadlock for process w.

3.3.2 On Receiving a Query

If a given query has been "seen before" at a process then a reply is generated, an action we call reflection. Queries not "seen before" cause new queries to be generated, an action called extension. Suppose an arbitrary process v receives query Q(B,w). Process v then searches its IQ-list for an entry Q(T,s) such that T $\propto$ B. There are two outcomes of the search:

1. { This action is called reflection. }
   There is some Q(T,s) that satisfies the search. Process v takes the following action:
   Send R(B,v) to process w;

2. { This action is called extension. }
   The search for Q(T,s) fails. First, Q(B,w) is added to process v's IQ-list. Suppose edges (v,x_1), (v,x_2), ... (v,x_K) exist in the PWFG. Process v now takes one of the following actions:

   a. {OR}

```
FOR j:= 1 TO k DO
   BEGIN
   Send Q(B,v) to x_j;
   Add Q(B,v) to the OQ-list;
   END;
```

   Notice that Q(B,v) may be added to the OQ-list numerous times. This is intentional: It is important that Q(B,v) appear k times in the OQ-list as a result of this step.

b. {AND}

```
FOR j:= 1 TO k DO
    BEGIN
    Send Q(B•<x_j>,v) to x_j;
    Add Q(B•<x_j>,v) to the OQ-list;
    END;
```

3.3.3 On Receiving a Grant

Section 2.6 specifies process behavior upon receiving grant messages. In this section, the rules are extended to accomodate the deadlock detection procedure. Suppose process v receives a grant from process w. In addition to the actions given in Section 2.6, one of the following steps is taken:

1. {OR} The OQ-list is made an empty list.

2. {AND} Entries of the form Q(T•<w>,v) are deleted from the OQ-list. That is, all record of queries sent to w is erased.

3.3.4 On Receiving a Reply

When a process receives a reply message, it is in response to some query sent earlier. The reply will be propagated only if no grant messages arrived since the query was sent, otherwise the reply is invalid. At an AND-requesting process, any valid reply causes immediate propagation. At an OR-requesting process, the reply propagation is delayed until all valid replies have arrived.

Suppose process v receives R(C,s). Then process v searches its OQ-list for an entry of the form Q(C,v). If the search fails, then no action is taken by process v, and the reply is ignored. Otherwise process v takes one of the following steps, which are collation actions:

1. {OR} The entry Q(C,v) is deleted from the OQ-list. After this deletion, Process v again searches its OQ-list for an entry of the form Q(C,v):

- If the search fails, then process v locates the entry Q(C,x) in process v's IQ-list. Process v deletes the entry Q(C,x) from the IQ-list, and sends R(C,v) to process x.

- If the search turns up some Q(C,v) in the OQ-list, then process v takes no further action.

2. {AND} The entry Q(C,v) is deleted from the OQ-list. Now process v searches its IQ-list for an entry Q(B,x) such that C = B•<s>:

   - If the search fails, then process v takes no further action, i.e., R(C,s) is ignored.

   - If the search found Q(B,x), then process v deletes Q(B,x) from the IQ-list, and sends R(B,v) to process x.

3.4 Example of Procedure Execution

In figure 4 is the PWFG used for this example. Process i is the initiator of the deadlock detection procedure. Process x has the only AND-request in the PWFG.

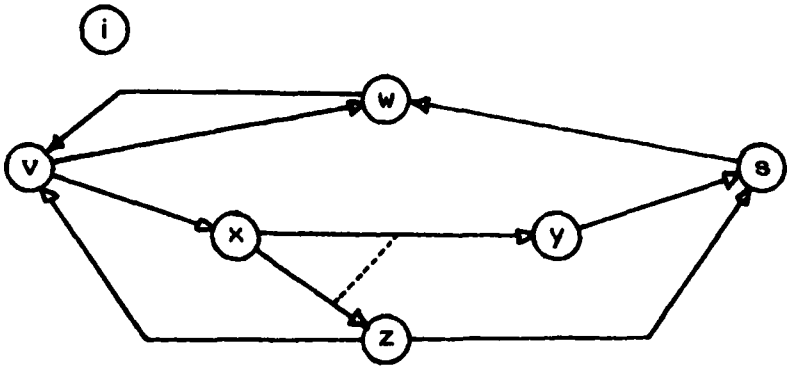


Figure 4. Example PWFG Showing Deadlock.

Figure 5 illustrates a trace of one possible execution of the procedure. Each element in the table shows an event for a process. All of the actions described within an event are considered to happen concurrently. The notation "Q(B,x)→y" means that query Q(B,x) is sent by process x to process y. "Q(B,x)•y" means that query Q(B,x) arrives at y. The sequence does not show the supporting updates to the IQ and OQ lists by the processes.

Event	Process v	w	x	y	z	s
1	Extension Q(<i>,i)•v Q(<i>,v)→x Q(<i>,v)→w					
2		Extension Q(<i>,v)•w Q(<i>,w)→v	Extension Q(<i>,v)•x Q(<iy>,x)→y Q(<iz>,x)→z			
3	Reflection Q(<i>,w)•v R(<i>,v)→w			Extension Q(<iy>,x)•y Q(<iy>,y)→s	Extension Q(<iz>,x)•z Q(<iz>,z)→s Q(<iz>,z)→v	
4	Reflection Q(<iz>,z)•v R(<iz>,v)→z	Collation R(<i>,v)•w R(<i>,w)→v				Extension Q(<iz>,z)•s Q(<iz>,s)→w
5	Collation R(<i>,w)•v	Extension Q(<iz>,s)•w Q(<iz>,w)→v			Collation R(<iz>,v)•z	Extension Q(<iy>,y)•s Q(<iy>,s)→w
6	Reflection Q(<iz>,w)•v R(<iz>,v)→w	Extension Q(<iy>,s)•w Q(<iy>,w)→v				
7	Reflection Q(<iy>,w)•v R(<iy>,v)→w	Collation R(<iz>,v)•w R(<iz>,w)→s				
8		Collation R(<iy>,v)•w R(<iy>,w)→s				Collation R(<iz>,w)•s R(<iz>,s)→z
9					Collation R(<iz>,s)•z R(<iz>,z)→x	Collation R(<iy>,w)•s R(<iy>,s)→y
10			Collation R(<iz>,z)•x R(<i>,x)→v	Collation R(<iy>,s)•y R(<iy>,y)→x		
11	Collation R(<i>,x)•v R(<i>,v)→i (Deadlock)		Ignored R(<iy>,y)•x			

Figure 5. Example Execution Sequence.

Notes on the sequence: In event 3, two messages are sent to process s: Q(<iy>,y)→s and Q(<iz>,z)→s. Because of the process model restriction on message arrival, process s receives only one message at a time. Accordingly, Q(<iy>,y)•s in event 5, and Q(<iz>,z)•s in event 4. Both messages cause extension at s because the queries have different labels.

After event 4, process w has experienced extension and collation (on behalf of process v's query) and w's IQ and OQ-lists are again empty. Then in event 5, process w again is extending, this time on behalf of Q(<iz>,s)•w.

Different cases of collation appear in this sequence. In event 4, collation at w is a case of one reply triggering another reply. Event 5, collation at z, is a case of a reply causing update to an OQ-list, but no further message action. Event 11, "ignored" at x, is explained in Section 3.3.4.

Here is a list of the contents of the IQ and OQ-lists after event 5 and before event 6. The "*" indicates a message that is also en route between events.

Process v	
---IQ----	---OQ----
Q(<i>,i)	Q(<i>,v)

Process w	
---IQ----	---OQ----
Q(<iz>,s)	Q(<iz>,w)*

Process x	
---IQ----	---OQ----
Q(<i>,v)	Q(<iy>,x)
	Q(<iz>,x)

Process y	
---IQ----	---OQ----
Q(<iy>,x)	Q(<iy>,y)

Process z	
---IQ----	---OQ----
Q(<iz>,x)	Q(<iz>,z)

Process s	
---IQ----	---OQ----
Q(<iz>,z)	Q(<iz>,s)
Q(<iy>,y)	Q(<iy>,s)*

In event 6, process v will reflect Q(<iz>,w)•v because Q(<i>,i) is in v's IQ-list, and <i> ⊴ <iz>. Also in event 6, process w will extend

because $Q(\langle iy\rangle,s)=w$, and it is not the case that $\langle iz\rangle \trianglelefteq \langle iy\rangle$. Therefore $Q(\langle iy\rangle,s)$ will be added to process w's IQ-list and $Q(\langle iy\rangle,w)\rightarrow v$.

4.0 IMPLEMENTATION ISSUES

The following sections show how the basic deadlock detection procedure can be enhanced. Section 4.1 suggests how initiation of the procedure should be controlled. In section 4.2 some concurrency restrictions are given. Sections 4.3 and 4.4 offer efficiency improvements.

4.1 Initiation

Suppose that a process has been blocked continuously for some time T, where T is a performance parameter. That process is therefore suspected of being deadlocked, and the controller initiates the detection procedure for that process. There may be many processes that qualify as suspects, and the controller could initiate the detection scheme for each of the suspects. However, within each "deadlock component" of the PWFG, it is sufficient for a controller to find one of the deadlocked processes, and it is desirable to limit the number of initiations to reduce message traffic. The controller should use a performance parameter K to limit the rate of initiations of the detection procedure. If this rate is too large, then much of the message traffic will be redundant, because the PWFG will not be changing faster than the detection procedure executes.

There is no conflict for several initiators to concurrently attempt to detect deadlock: Processes serve any number of deadlock detection computations by maintaining the IQ and OQ-lists. If an execution of the procedure does not detect deadlock, one result is that, for some processes, the IQ-list and OQ-list will retain useless entries. The following scheme will clean up the IQ and OQ-lists. Suppose some initiator is created to detect deadlock for suspected process w. Let the new initiator be named w_K, which means initiator for process w, version k. The previous (if any) initiator for process w was named w_{K-1},

and successive initiators will have increasing version numbers. Then each time a process receives a query, the obsolete IQ and OQ-list entries can be expunged, for they all have labels of the form $\langle w_j \ldots \rangle$, $j < k$.

4.2 Concurrency of Process Execution

Concurrency of processing at different sites is expected for a distributed program. The procedure in this paper also permits concurrent execution within a site, subject to the following restriction: Each process must receive a message and execute an action (reflection, collation, extension) sequentially. This restriction maintains the integrity of the IQ and OQ-lists, as well as insuring that a process receives only one message at a time. Interleaving the actions of different processes, and parallel execution of different processes are allowed.

4.3 Sharing Deadlock Status

The procedure may be improved by allowing processes to propagate deadlock status. Once a controller makes the determination that a process is in deadlock, all queries sent to that process may be immediately reflected (all queries would eventually be reflected anyway). This modification improves the performance of the detection procedure under concurrent initiations of the procedure by different processes.

4.4 Reducing Message Traffic

Processes communicate without regard to site location. The execution of the detection procedure amounts to controllers sending and receiving many small messages. In order to reduce the message traffic, controllers may elect to "batch" communication, i.e., retain process messages and accumulate packets. This consideration may be automatically handled as part of the communication protocol, as long as the order of messages is log-

ically preserved for inter-process communication (c.f. Section 2.2).

Controllers should give priority to message traffic within a site. These are internal operations for a controller. By preferring the internal message traffic, the controller will detect local deadlocks without outside communication.

5.0 SUMMARY

We have presented a scheme to detect deadlock in distributed systems. A sophisticated model of resource requests was used. The structure of the procedure is distributed, and the granularity of the distribution is of the same order as the resource requests. The scheme is dynamic, and does not require that graph structures be maintained. The procedure specifies that controllers concurrently execute the detection procedure, and that concurrency within a controller is permitted. The procedure is not susceptible to "false deadlock" detection, and we prove that the procedure is correct.

6.0 APPENDIX: PROOF OF DEADLOCK DETECTION PROCEDURE

The goal of the proof is defined in Section 6.1, and a local definition of deadlock is specified. Some auxiliary definitions are found in Section 6.2. As well as supporting the proof, these definitions are useful intuitive bases for understanding the procedure. The subject of Section 6.3 is the termination of a tree computation. Section 6.4 assigns a semantic meaning to reply messages and proves that this meaning is upheld by the detection procedure. In Section 6.5 the proof is completed by verifying correctness conditions.

6.1 Criteria for Correctness

The deadlock detection procedure is not an algorithm. That is, the deadlock situtation will be detected, if it exists, but the procedure will not detect "no deadlock" situations. Correctness means that the deadlock detection procedure must satisfy the following conditions:

P0 — If process w is deadlocked prior to v's initiation of the detection procedure, then process v will detect deadlock for process w after a finite time.

P1 — If process v detects deadlock, then a deadlock situation truly exists.

P1 is a partial correctness condition, and P0 is a termination condition.

For the purpose of this proof, the following local definition of deadlock is employed: Process v is deadlocked iff it is permanently blocked in the PWFG. More precisely:

Definition: Process v is deadlocked iff

- Process v has an AND-request, has sent a request for each outgoing edge, but for at least one of these edges, no grant will ever be received by process v.

- Process v has an OR-request, has sent a request for each outgoing edge, but no grant will ever be received by process v.

We call this a local definition because it does not contain the definition of the underlying static graph structure in the PWFG that causes deadlock. The local definition also includes the possibility of a process permanently blocked due to starvation, or because another process is in an infinite loop. We exclude these possibilities from consideration: They are beyond the scope of deadlock detection.

6.2 Auxiliary Definitions

6.2.1 Tree Computation

A tree computation is a distributed computation. The tree computation grows by sending queries and shrinks by receiving replies. When a tree computation shrinks back to its root, it terminates. The deadlock detection procedure uses tree computations to search for active processes in the PWFG. The paradigm for this type of distributed computation is due to Dijkstra and Scholten [8].

In the deadlock detection procedure, the tree computation has an additional aspect. It can be viewed as a distributed data structure. The tree computation consists of the set of all IQ and OQ-list elements having a common label.

Each tree computation is uniquely associated with a label. A query or reply's label identifies a tree computation. When a label is first generated by a process then a tree computation is created. Creation of a tree computation happens in two ways: When a process initiates the detection procedure, and when extension occurs at an AND-requesting process. The latter event causes new, offspring trees to be generated. The next section formalizes this relationship.

6.2.2 Label Descendancy

A descendancy relation is defined for labels. Label B is descended from, or equal to, label C iff C $\preceq$ B. Instead of the prefix operator, the notation B$\preceq$C will be used to mean that B is descended from or equal to C. Note that all labels for an execution of the detection procedure have a common ancestor, the root label, which identifies the initiator of the detection procedure. Given labels B and C, any process can determine whether or not B$\preceq$C.

Since labels identify tree computations, and queries and replies have labels, we will also say that tree computations are related by descendancy.

6.3 Termination of Tree Computations

Tree computation T terminates when v, the process that created T, obtains a reply for each query that v sent with T's label, with no intervening grants. That is, between the sending of a query Q(T,v) along edge (v,w), and the receiving of reply R(T,w), v did not receive a grant from w.

6.3.1 Tree Computation Termination Lemma

Tree computation T terminates iff for every x and y such that Q(T,x) is sent to y, then R(T,y) arrives at x with no intervening grants.

Proof: First we show that non-termination $\Rightarrow$ (intervening grants or missing replies): Suppose T

will not terminate. Then v_0, the creator of T, sent some queries $Q(T,v_0)$, and

1. v_0 will not obtain all replies corresponding to the queries, or

2. v_0 receives a grant on some edge (v_0,v_1) before $R(T,v_1)$ is received, although all replies are received by v_0.

This is one half of the desired equivalence.

Second, we show that termination ⇒ (no intervening grants or missing replies): Now suppose T terminates. Then v_0, the creator of T, sent some set of queries $Q(T,v_0)$, and obtained corresponding replies with no intervening grants, by definition of T's termination. Consider some arbitrary process v_1 that did not reflect v_0's query to v_1 (reflection trivially satisfies the result). v_1 therefore collated to produce a reply $R(T,v_1)$ sent to v_0. If v_1 had an AND-request, no additional queries on T's behalf were sent, and the result is complete. Otherwise v_1 had an OR-request, and replied to v_0 because it sent some queries $Q(T,v_1)$ and obtained corresponding replies. Moreover, if a grant was received on edge (v_1,v_J) between the sending of $Q(T,v_1)$ and receiving $R(T,v_J)$, then collation would not have occurred (grants diminish the OQ-list and cause replies to be ignored). An intervening grant is ruled out.

Now consider some arbitrary process v_2 that did not reflect v_1's query to v_2. The argument may be repeated until some process v_K is reached, and v_K obtained all of its replies by reflection. Since the path v_0, v_1, v_2, ... v_K was arbitrarily chosen, all queries did lead to replies for tree computation T, with no intervening grants. ■

6.4 The Meaning of Reply

The following assertion can be made at the point when and where process v sends reply R(T,v) to

process w in response to Q(T,w), previously sent from w to v:

1. w will never receive (or have received) a grant from v after w sent Q(T,w) to v, or

2. Tree computation T, or some ancestor of T, will not terminate.

As shorthand for this rather complicated assertion, the notation {v\w\T} will be used. Using the notation of Section 3.4,

L0 $R(T,v)\rightarrow w \Rightarrow \{v\backslash w\backslash T\}$

is an invariant assertion for the deadlock detection procedure. Before we prove this invariant in the next section, some smaller results are given below.

L1 $R(T,v)\blacksquare w \Rightarrow \{v\backslash w\backslash T\}$

L1 follows from L0, because $R(T,v)\rightarrow w$ must be in response to some Q(T,w) sent previously from w to v. If a grant was received by w from v between $Q(T,w)\rightarrow v$ and $R(T,v)\blacksquare w$ then T will not terminate by the tree computation termination lemma. L0 asserts that no later grant will be forthcoming unless T (or ancestor) fails to terminate, so L1 is implied.

L2 $R(T,v)\blacksquare w$ and w has AND request $\Rightarrow \{v\backslash w\backslash U\}$, where $T \subset U$.

$R(T,v)\blacksquare w$ is in response to $Q(T,w)\rightarrow v$. If, in between these two events, w received a grant from v, then T will not terminate. Otherwise T must terminate because w created T exclusively for edge (w,v). By L1, {v\w\T} holds, but now that T has terminated the assertion is {v\w\U}, where U is the parent of T.

6.4.1 Verification of Invariant

Replies are generated by reflection and collation. Collation will first be considered.

6.4.1.1 Collation

Suppose process v is about to send process w a reply, R(T,v). We wish to show {v\w\T}. There must have been a query, Q(T,w), previously sent from w to v. There are two cases to consider:

1. v has an AND-request. Since v is collating to produce a reply, v must have sent a query Q(S,v) to some process z, and subsequently obtained a reply R(S,z) from z, where S⊏T. There cannot have been any grant from z to v between the sending of Q(S,v) and receiving of R(S,z), or else the collation would not occur. By L2, R(S,z)∎v ⇒ {z\v\T}, because T is the immediate ancestor of S.

   In turn, v will not send a grant to w until v gets a grant from z (recall v has an AND-request). This would imply non-termination of T or some ancestor of T because of {z\v\T}. If v has already sent a grant to w then it will arrive before R(T,v) and T will not terminate due to the tree computation termination lemma. It is safe to assert {v\w\T}.

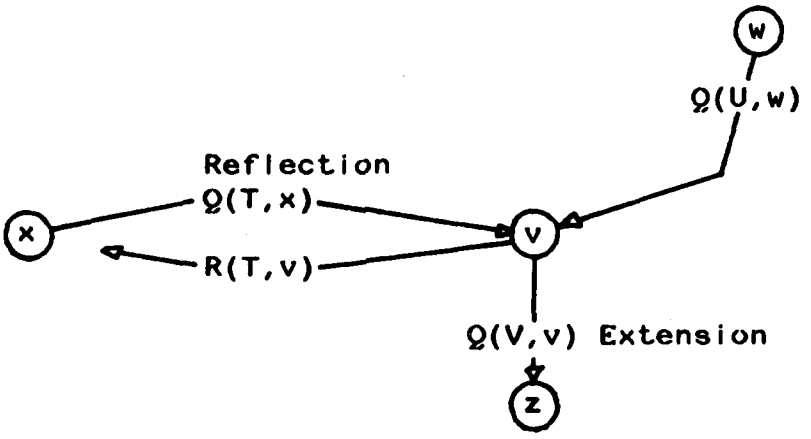
2. v is an OR vertex. Since v is collating to produce a reply, v must have sent queries Q(T,v) to some processes z_1, z_2, ... z_n, and subsequently received replies $R(T,z_1)$, $R(T,z_2)$, ... $R(T,z_n)$. Any grant from a process z_J cannot have been received at process v between the sending of Q(T,v) and receiving $R(T,z_J)$, or collation would not occur (this also rules out cancellation of any (v,z_K)). v will not send a grant to w unless v receives a grant from some z_K. But $\{z_K\backslash v\backslash T\}$ holds because $R(T,z_K)$ has been received, with no

intervening grants, so it is safe to assert {v\w\T}. ■

6.4.1.2 Reflection

Suppose process v is about to sent R(T,v) to process x by reflection. We wish to show {v\x\T}. There must have been a query Q(T,x) sent by process x. This is the situation (see figure 6):

Process w sent query Q(U,w) to process v. Process v did not reflect Q(U,w), and has not yet sent a reply R(U,v) to w. Now v has received query Q(T,x), which is to be reflected because T⊑U.



Note: T⊑U and V⊑U; First extension occurred at v, and now reflection is to take place.

Figure 6. Situation for Reflection.

If process x receives a grant from v before R(T,v), then T will not terminate. In order to send a grant to x after v sends R(T,v), v must become active. There are two cases:

1. v becomes active before sending R(U,v) to w. In this case U will not terminate.

2. v becomes active after sending R(U,v) to w. But this implies that v received a reply, R(V,z) from some process z before a grant from z was received by v. The assertion {z\v\V}, and the grant from z now imply that U, or some ancestor of U, will not terminate.

In either case, it is safe to assert {v\x\T}. ■

6.5 Verification of Correctness Conditions

6.5.1 Verification of P1

Suppose process i is the initiator and declares deadlock for process v as the result of collation that produced $R(T,v) \rightarrow i$. Any reply $R(S,z_K) \blacksquare v$ that led to v's collation implies $\{z_K \backslash v \backslash S\}$, where $S \subseteq T$. But S has terminated, T has terminated, and T has no ancestors. z_K will therefore never send a grant to v. The conclusion is that v is truly deadlocked. ■

6.5.2 Verification of P0

We verify P0 by assuming that the deadlock detection procedure will fail to detect deadlock and proving a contradiction by construction.

Suppose process v_θ is deadlocked and receives an initiator's query, but deadlock will never be detected. Consider two cases:

1. If v_θ has an AND-request, numerous tree computations were created, but none of them will terminate because, by hypothesis, deadlock will never be detected. It is possible that a tree computation will not terminate because v_θ received a grant between the times of sending a query and getting a reply; this cannot be true of all of v_θ's tree computations -- the deadlock assumption would be violated. v_θ therefore sent some query $Q(T_\theta, v_\theta)$ to v_1, and will never receive a message from v_1.

2. If v_0 has an OR-request, one tree computation was created. Under the deadlock assumption and the non-detection hypothesis, v_0 sent some query $Q(T_0,v_0)$ to v_1 and will never receive a message from v_1.

In both cases, it is asserted that some v_1 will send neither a grant nor $R(T_0,v_1)$ to v_0.

Now consider v_1, which received $Q(T_0,v_0)$ after a finite time and will will never send $R(T_0,v_1)$ to v_0. v_1 cannot be active, since activity might lead to v_1 sending v_0 a grant. That is, with this deadlock detection procedure, starvation will not be detected. There are now two cases for v_1:

1. v_1 has an AND-request, and will never collate to produce a reply for v_0. v_1 therefore sent some query $Q(T_1,v_1)$ to some v_2 and v_1 never received $R(T_1,v_2)$. We may also assert that v_1 never gets a grant from v_2, by an appropriate choice of v_2. Contradiction of this assertion violates the hypothesis. $T_1 \sqsubset T_0$.

2. v_1 has an OR-request and will never collate to produce a reply for v_0. v_1 therefore sent some query $Q(T_1,v_1)$ to v_2, and v_2 will never send $R(T_1,v_2)$, nor will v_2 send v_1 a grant, by hypothesis. $T_1=T_0$.

In both cases, it is asserted that some v_2 will send neither a grant nor $R(T_1,v_2)$ to v_1, $T_1 \sqsubseteq T_0$.

Now consider v_2, which received $Q(T_1,v_1)$ after a finite time and will will never send $R(T_1,v_2)$ to v_1. v_2 cannot be active, since activity might lead to v_2 sending v_1 a grant. Nor can v_2 be part of T_1 (or any ancestor of T_1), because reflection would occur and v_1 would get a reply in finite time. Therefore, v_2 will not reply to v_1 because collation did not occur at v_2. The case argument for v_2 is repeated, as above.

Eventually some v_K will be reached, and all of v_K's queries must be reflected, simply because the

number of processes is finite. If v_K will never send a reply, then it cannot be because of grants interfering with the reflection of v_K's queries, for then deadlock would be contradicted by the choice of v_1, v_2, ... , v_K, and hypothesis. But all of v_K's queries will be reflected in finite time, so v_K will obtain replies in finite time. Therefore v_K will collate to reply in finite time, but this violates the assumption about v_K. Contradiction. ■

7.0 REFERENCES

[1] Ho, G. S., and Ramamoorthy, C. V. "Protocols for Deadlock Detection in Distributed Database Systems." IEEE Transactions on Software Engineering SE-8, 6(Nov 1982), 554-557.

[2] Menasce, D. A., and Muntz, R. R. "Locking and Deadlock Detection in Distributed Data Bases." IEEE Transactions on Software Engineering SE-5, 3(May 1979), 195-202.

[3] Obermarck, R. "Distributed Deadlock Detection Algorithm." ACM Transactions on Database Systems 7, 2(Jun 1982), 187-208.

[4] Beeri, C., and Obermarck, R. "A Resource Class Independent Deadlock Detection Algorithm." Research Report RJ3077(38123), IBM Research Laboratory, San Jose, California, May, 1981.

[5] Gligor, V. D., and Shattuck, S. H. "On Deadlock Detection in Distributed Systems." IEEE Transactions on Software Engineering SE-6, 5(Sep 1980), 435-439.

[6] Chandy, K. M., and Misra, J. "A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems." in Proceedings of ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing, Aug(1982).

[7] Chandy. K. M., Misra, J., and Haas, L. M. "A Distributed Deadlock Detection Algorithm and Its Correctness Proof." to appear in ACM Transactions on Computing Systems, (May 1983).

[8] Dijkstra, E. W. D., and Scholten, C. S. "Termination Detection for Diffusing Computations." Information Processing Letters 11, 1(Aug 1980), 1-4.

[9] Lamport, L. "Time, Clocks and the Ordering of Events in a Distributed System." Communications of the ACM 21, 7(Jul 1978).

[10] Holt, R. C. "Some Deadlock Properties of Computer Systems." Computing Surveys 4, 3(Sep 1972) 179-196.

END

FILMED

1-84

DTIC