MIT/LCS/TR-185

DEADLOCK DETECTION IN COMPUTER NETWORKS

Barry Goldman

September 1977

MIT/LCS/TR-185

DEADLOCK DETECTION IN COMPUTER NETWORKS

Barry Goldman

September 1977

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LABORATORY FOR COMPUTER SCIENCE
(formerly PROJECT MAC)

CAMBRIDGE MASSACHUSETTS 02139

Deadlock Detection in Computer Networks

by

Barry Goldman

Submitted to the Department of Electrical Engineering and Computer Science on March 1, 1977, in partial fulfillment of the requirements for the Degrees of Bachelor of Science and Master of Science.

ABSTRACT

The problem of detecting process deadlocks is common to transaction oriented computer systems which allow data sharing. Several good algorithms exist for detecting process deadlocks in a single location facility. However, the deadlock detection problem becomes more complex in a geographically distributed computer network due to the fact that all the information needed to detect a deadlock is not necessarily available in a single node, and communications delays may lead to synchronization problems in getting an accurate view of the network state.

In this Thesis, two published algorithms dealing with deadlock detection in computer networks are discussed, and examples demonstrating the failure of these algorithms are given. Two algorithms are then presented for detecting deadlocks in a computer network which allows processes to wait for 1) access to a portion of a database, or 2) a message from another process. The first algorithm presented is based on the premise that there is one control node in the network, and this node has primary responsibility for detecting process deadlocks. The second, and recommended, algorithm distributes the responsibility for detecting deadlocks among the nodes in which the involved processes and resources reside. Thus a failure of any single node has limited effect upon the other nodes in the network. A computer model of the "decentralized" (second) algorithm was designed and it is described in the Thesis.

THESIS SUPERVISOR: Stephen A. Ward
TITLE: Assistant Professor of Computer Science and Engineering

Acknowledgements

As a participant in M.I.T.'s VI-A (Electrical Engineering and Computer Science Co-operative) Program, I was able to write this Thesis based on research that I conducted while working in the Advanced Systems Engineering Group of Honeywell Information Systems, Inc. (Billerica, Mass.) I would like to thank Charles W. Bachman, who acted as my supervisor at Honeywell. He suggested the Thesis topic, and gave me valuable advice throughout various phases of the project. I would also like to thank others in the group for the help they gave me in conducting the research and in the writing of the Thesis. They are Mike Canepa, William Helgeson, Beth Lang, Maxine Neil, Charlotte Reiley, Mario Trinchieri and Paul Wood.

Additionally, thanks go to Steven Taylor, who provided me with some feedback in the early stages of my research after introducing me to Mr. Bachman. Finally, I would like to thank Professor Stephen Ward for his work in the supervision of the Thesis, and John Tucker and Lydia Wereminski for running the VI-A Program.

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract No. N00014-75-C-0661.

TABLE OF CONTENTS

I. Introduction

A simple example of deadlock (or "deadly embrace") occurs when a process P1 is blocked while waiting for access to resource R2 which is controlled by process P2, and P2 in turn is blocked while waiting for access to resource R1 which is controlled by P1. A deadlock may involve more than two processes. For example, process P1 may be waiting for access to resource R2 which is controlled by process P2, P2 may be waiting for access to resource R3 which is controlled by process P3, ..., process P[n-1] may be waiting for access to resource Rn which is controlled by process Pn, and Pn may be waiting for access to resource R1 which is controlled by P1.

Multiprocessing and data sharing are commonly used in a single location transaction oriented computer system. In the future they will be common to transaction oriented, geographically distributed computer networks. In this Thesis an algorithm is presented that can be used to detect deadlocks involving processes waiting for access to a shared portion of a database or waiting for a message from a process with which it is communicating within a computer network. It is possible that a process can be either computerized or manual, although a manual process (i.e. a person at a terminal) can not directly request access to a portion of a database, as it is restricted to only communicate with computerized processes by the use of messages. Throughout this paper, the word "operator" will be used to refer to a manual process.

Much has been written dealing with deadlock detection, avoidance and prevention in computer systems. However, most of the literature discusses a single location facility where the status of all processes and resources are available in a single local table. (For a good discussion, including a graph model of computer systems which can be used to detect deadlocks, see "Some Deadlock Properties of Computer Systems" [7].) Very few articles have been published that are concerned with the deadlock problem in a computer network (geographically distributed computer system).

When dealing with a computer network as opposed to a single location facility, the deadlock detection problem becomes more difficult due to the fact that all the information needed to detect a deadlock is not necessarily available in a single node, and communication delays may lead to synchronization problems in getting an accurate view of the network state. Some reasons for restricting access to portions of a database (even though the result of blocking processes can lead to deadlock) and some reasons why the common deadlock prevention and avoidance algorithms are not well suited to the networks under consideration will be discussed. Several deadlock detection schemes for computer networks (some from recent literature, some designed by this author) will be presented, and they will be followed by a discussion of some of the benefits of using the various schemes.

I.1 The Interference Problem

Given two or more independent processes, interference is said to have occurred if the results produced by their concurrent execution would not have been obtained by running these processes one at a time in any order (i.e. nonconcurrently).

A simple example of interference is the following. Let two processes, P1 and P2, read the contents of database record R1. Then let P1 add 5 to the value and let P2 add 10 to the same value. Now let each process alter the contents of R1 to contain the value computed by that process. Depending upon the order of update, the contents of R1 will be either 5 or 10 greater than the value that was contained during the reads. We have a case of interference because the value of R1 would have been 15 greater than the value contained at the time of the first read if P1 and P2 had been executed sequentially in either order.

Another case of interference occurs when a process, in processing one transaction, twice alters the contents of the same database object and in between the two writes, a second process reads the contents of that database object. In some cases a process which is only reading the contents of a database object may not care if there is any interference, in which case it may request "dirty read" access to the database object. (A process that is only reading the contents of a database object can not interfere with the values produced by another process, although other processes can interfere with the values produced by the "reading" process.)

When maximum concurrency among independent processes is desired, a process must be allowed to read and alter the contents of a database object whenever it wants to. (This type of access to data has been called "shared read/shared write".) In order to detect interference, records must be kept about the type of use (read or write) of each database object, and what processes (and when they) used it. An algorithm to detect interference when this information is kept is presented in "On Managing Interference Caused by Database Sharing" [10]. A more thorough discussion of interference is also given. After an interference situation is detected, at least one of the involved processes must be forced to rollback to a previous state in order to correct the interference condition.

Most systems, in order to avoid interference and guarantee that a process will see a consistent state of a database, restrict access to data by a system of locks. If a process wants to change the contents of a database object, it must request exclusive access to that database object, thus temporarily (for the duration of the lock) preventing all other processes from accessing that database object. If a process only wants to read the contents of a database object, it can request shared read access to that database object, thus temporarily (for the duration of the lock) preventing all other processes from altering the contents of that database object. If a database object can be shared among several readers, the method of access is called "shared read/exclusive write", whereas if there can be only one reader, it is called "exclusive read/exclusive write".

When a request for access to a database object (resource) can not be granted due to the existence of a lock on that database object, the requesting process must be blocked until the resource becomes available. Due to processes waiting for access to resources, there exists the possibility of deadlock among the processes in a computer system.

I.2 Deadlock Prevention

Deadlock prevention schemes place constraints upon system users in order to ensure that deadlock will never occur. There is little operating system overhead involved when using prevention methods. There are several deadlock prevention algorithms that are widely known:

1. Each process must request all needed resources at one time and will remain blocked until all requests can be granted simultaneously. (This is often referred to as "static" allocation.)

2. All resources are given a unique number and processes must request resources, one at a time, in numerical order.

3. When an active process requests a resource that is controlled by a blocked process, the blocked process must release the resource so that it may be allocated to the active process. A process will go from the active to blocked state only if it requests a resource controlled by another active process.

The unpredictability of resource usage in a transaction oriented system, plus the loss of productivity that results from tying up resources unnecessarily or forcing processes to release resources and request them later (which often results in some redundant computations due to a process having to repeat some operations to maintain a consistent database) make prevention algorithms undesirable for use in the systems under consideration. In a multiprocessing environment which considers inter-process messages as resources, it is impossible to have an advance knowledge of all the resources that will be needed by a process. Thus algorithm 1 can not be used in this type of system, whether it is a single or multi node facility. Algorithm 2 is unsuitable for the systems under consideration because although it may be possible to give a unique number to each inter-process message, a process must be "allocated" each message that it will send to another process, which can result in many difficulties when two processes are sending several messages to each other. Algorithm 3 can not be used because it implies that all resources must be pre-emptable (i.e. they must be able to be released by a process upon the demand of the system), which is an impossible situation when messages are treated as resources.

I.3 Deadlock Avoidance

Deadlock avoidance algorithms calculate safe paths for completion of all processes. Before a resource is allocated to a given process, the operating system checks if there would be at least one path via which all processes can run to completion after the allocation is made. If no such path exists, then the requesting process must wait until a time when the resource can be safely allocated to the process. Avoidance algorithms thus force processes to wait unnecessarily in order to be certain that all processes will be able to run to completion without the threat of deadlock.

In "System Deadlocks" [5] it is stated that "to avoid deadlocks in a multiprogramming system in which the necessary conditions for deadlocks can exist, it is usually necessary to have some advance information on the resource usage of tasks." When portions of databases are considered resources, and they are locked at a level lower than a file (page, record, field, etc.), it is difficult to determine in advance what database objects will be needed. In addition, due to the unpredictability of processes in a transaction oriented system, it is impossible to have an advance knowledge of all the inter-process messages that will be requested by a process. Therefore, deadlock avoidance algorithms can not be used in a single or multi node transaction system which permits inter-process communication.

I.4 Deadlock Detection

Since it seems that deadlock prevention and avoidance algorithms are unsuitable for the distributed systems under consideration, deadlock detection methods must be examined. When employing a deadlock detection algorithm, requested resources are usually assigned to the requesting processes whenever possible, and processes are blocked only when desired resources are unavailable. Either the operating system or a system user must occasionally check for a deadlock situation, and if one is found, must rollback (backup) and retry at least one process in order to break the deadlock. (It is hoped this will force a new sequence of access to resources.)

From the implementor's viewpoint, the easiest strategy to adopt is that where one assumes deadlock occurs infrequently. In this case someone (an operator) external to the network would have the responsibility for detecting the deadlock and deciding what process should be forced to rollback to a previous state. With this approach the only overhead involves the temporary inability to access the resources controlled by the deadlocked processes and the cost of rollback/retry of some (or all) of the deadlocked processes. (This cost may be large for each deadlock, but if there are few deadlocks the overall system cost may be less than it would be if there were a "deadlock detector" that was constantly checking for deadlocks.) One could also assume that if a process has been blocked for 'X' units of time, then it is deadlocked and the operating system should force it to rollback to a previous state, although this strategy may result in some unnecessary redundant computations because some processes that will be retried may not have been involved in a deadlock.

At least two articles have been published which propose protocols for allocating database objects in a computer network in a manner such that deadlock can be detected at the time a request for access is denied. In designing an algorithm to be used to detect process deadlocks in a transaction oriented computer network which allows process to process communication, it is necessary to allow for the possibility of a process waiting for a message from another process (which may be manual or computerized). Additionally, a process must be allowed to wait for access to a database object which has been allocated to at least one other process.

Any algorithm that will be implemented as part of an operating system should be as efficient as possible. Therefore, in the algorithms proposed by this author, an attempt was made to minimize the number and size of internodal messages involved in the detection of deadlocks.

I.5 Structure of the Thesis

Chapters II and III contain descriptions and comments (including some examples pointing out deficiencies) relating to two papers that have been published proposing protocols for allocating database objects in a computer network such that deadlock can be detected at the time a request for access to a database object is denied. Chapter IV presents an introduction to the two schemes for detecting deadlock in a computer network that are proposed by this author in Chapters V and VI. The two schemes differ in that one (Chapter V) places the primary responsibility for detecting deadlock anywhere in the network on one control node, whereas the other totally distributes the responsibility throughout the network. Chapter VII contains a discussion of a functional model of the algorithm proposed in Chapter VI. The Appendices contain a description and demonstration of the model, in addition to containing the PL/I code for the model itself. Chapter VIII contains some suggestions for future research, and Chapter IX contains a comparison of the various algorithms presented in Chapters II, III, V and VI, plus some concluding remarks.

If one only wants to read about the algorithm that is recommended by this author, it is possible to read Chapters IV and VI with no loss of understanding. Chapter VII can also be understood after reading Chapters IV and VI; as can the Appendices and some portions of Chapter VIII.

II. Proposal of Chandra, Howe and Karp

In "Communication Protocol for Deadlock Detection in Computer Networks" [3], a scheme is presented which the authors call "a novel solution to the deadlock problem in the network environment." Their "solution" is described below, and the description is followed by an example where the scheme allows a deadlock to go undetected.

II.1 Chandra, Howe and Karp's Proposed Solution

The authors propose that each installation (node) maintain a resource table (RT) which contains information about which processes have been allocated local resources, which processes have been queued (waiting for access) for local resources, which local processes have been allocated remote resources and which local processes have been queued for remote resources. The type of access requested by each process is also recorded. The authors claim that in a single node facility, there are several well known algorithms for detecting deadlocks using the tables mentioned above. They then state "it is believed to be obvious that these same algorithms would suffice in the multiple installation case provided that the resource table were to be expanded to include the pertinent information from the remote sites." A scheme to expand the resource table in a node is given in the paper.

The authors believe there are three types of requests for resources that can lead to deadlock. (In all cases, "it is assumed that the requested resource is not available, because, if it were, the allocation would take place immediately.") The action taken for each type of request is the following (as stated in the paper):

Case 1

A process requests a local resource, which is allocated to a local process, and all of the processes which are queued for this resource are also local processes. All of the necessary information is contained in the local RT, and the request is resolved locally.

Case 2

A process requests a local resource, which is either allocated to a remote process or one or more of the processes that are queued for this resource are remote processes. In this case, all of the RT's must be obtained by the local installation since deadlock may occur. Once all of the RT's have been obtained, the deadlock-determination algorithm can be applied to the expanded RT which contains all of the resources and processes in the total community of installations.

Case 3

A process requests a resource at some remote installation. In this case, the requesting installation forwards the request and its RT to the installation which has the requested resource. This installation then determines if the request can be honored immediately or if all of the RT's must be first obtained. In the case where the requested resource is allocated to or queued by only processes local to the two involved systems, the request can be honored immediately. Otherwise, this installation obtains the RT's from the remaining installations and then resolves the request.

In all of these cases, the RT's that are involved in the decision procedure must be locked until after the decision has been made. If the decision involves the RT's of the other installations in the community, these installations must be notified after the decision is made and their table is then released. In Case 3, the updates to the RT must be returned to the requesting installation while all other tables can be discarded and a simple release notice returned.

A description is given of the actions to be taken when "two or more installations may simultaneously request the various RT's in order to make an allocation for two or more independent requests."

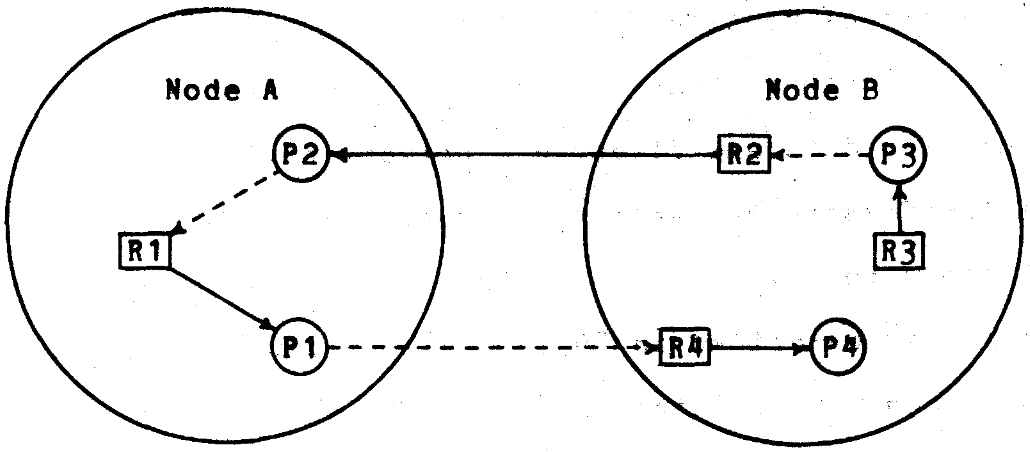
II.2 A Fault in the Proposed Solution

There are some resource requests which fall under Case 1, and result in a deadlock for which the local RT does not contain enough information to detect. Consider the following example:

> Let the network consist of two nodes, A and B. Let processes P1 and P2 and resource R1 be local to A, and let processes P3 and P4 and resources R2, R3, and R4 be local to B. Assume the following state of the network. (Figure II.1a contains a diagram of this "intermediate" state.) P1 has exclusive control of R1 and is queued waiting for access to R4, P2 has exclusive control of R2 and is queued waiting for access to R1, P3 has exclusive control of R3 and is queued waiting for access to R2, and P4 is active (non-blocked) and has exclusive control of R4. In this state there is no deadlock. Now let P4 request access to R3 and be queued for the resource. A deadlock now exists (see Figure II.1b) involving all four processes and all four resources. With the tables as described in the article, this deadlock could not be detected unless node A sent node B its tables, but this does not take place because the request falls into Case 1 (since P4 is local to B, as are P3

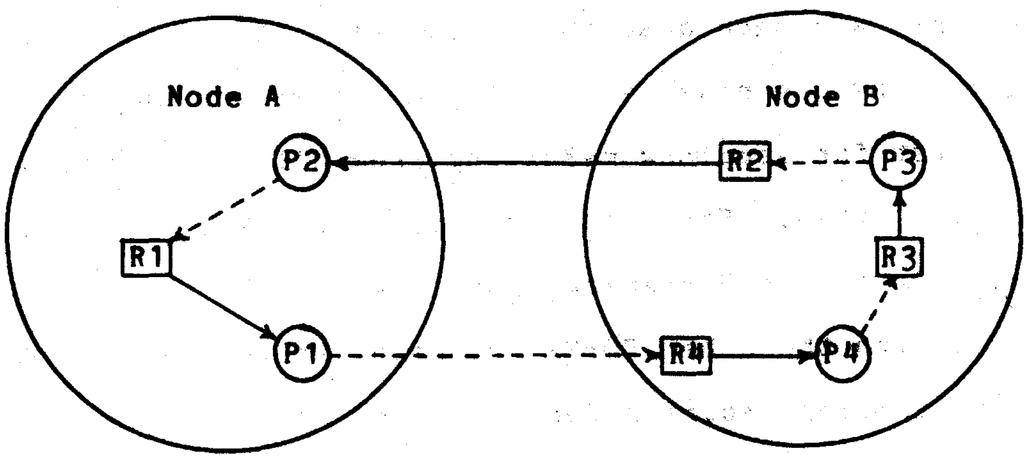
and R3). Therefore the deadlock goes undetected.

Similar examples (for networks consisting of three or more nodes) exist where requests falling under Case 3 result in undetected deadlocks. RT's from 3 or more nodes may be needed even if "the requested resource is allocated to or queued by only processes local to the two involved" nodes.



Intermediate State Diagram

Figure II.1a



Final State Diagram

Figure II.1b

KEY

Represents a process

Represents a resource

Represents a process having exclusive use of a resource

Represents a process waiting for access to a resource

III. Proposals of Mahmoud and Riordon

In "Protocol Considerations for Software Controlled Access Methods in Distributed Data Bases" [8], two schemes are presented for allocating database files in a network environment. The authors (Professors at Carleton University, Ottawa, Ontario, Canada) claim that with their schemes, by using the graphic representation as described in [9], deadlocks can be detected at the time an allocation decision is made. The two schemes are described below, and a brief discussion about the schemes follows, including an example where one of the proposals allows a deadlock to go undetected.

The first approach described requires that all deadlock tests be made by one node, whereas with the second approach each node must test for deadlock resulting from different processes accessing its files. Each node in the network will contain a Distributed Data Base Management Facility (DDBMF) which will communicate with the other DDBMF processes in the network for the purpose of handling requests for local and remote processes.

III.1 Mahmoud and Riordon's Centralized Control Approach

In the centralized approach, one node, called the control node, will make all the deadlock tests and handle all file allocations. If a process running at node i would like access to a file in node j, a request is sent to the DDBMF in node i, which then relays it to the central DDBMF, even if node i and node j are the same. Since the central DDBMF makes all the file allocation decisions, it has an overall picture of the global network status, and can therefore decide if the request can safely (without deadlock) be placed on the file queue.

III.2 Mahmoud and Riordon's Distributed Control Approach

In the distributed approach, the DDBMF at each node will have full control over all access to the files located at its node. As a result of this, the authors state that "each node DDBMF will be responsible for handling job interference (deadlock) problems that may arise while different processes are accessing its files." In order to avoid or detect deadlocks involving processes and files located at two or more nodes, "each individual DDBMF must obtain information from other DDBMF processes indicating the status of their files and queue tables. The information will be used ... to construct a global picture of the network and thus enable each individual DDBMF process to make the correct decisions."

All active user processes are separated into two classes. In the authors' own words,

> The classification is based on the localities of the files requested by the process and the type of access to each of these files:
>
> Class 1: each process belonging to this class has the following properties:
> 1) All files accessed by the process during its active session are located in a single node.
> 2) All files being updated by the process are single-copy files in the network (i.e. only a single copy of each file exists in the network).
>
> Class 2: each user process belonging to this class has the following properties:
> 1) Files that are accessed simultaneously by the process

during an active session do not all exist in a single computer system and/or

2) Any one of the files being updated by the process has multiple copies in the network.

It is obvious that the two classes of processes are mutually exclusive.

The authors suggest using a graph representation in order to detect deadlock, and they describe how a DDBMF gets information from the other DDBMF's in the network and when it should check for deadlock:

> Assume that there are n nodes in the network, i.e., n individual DDBMF processes. Each process will transmit (n-1) identical messages simultaneously, with one message addressed to each of the remaining DDBMF processes. Each message contains the most updated information about the status and queues of files at the node in question. The messages will be transmitted periodically at the onset of synchronous clock intervals. Similarly, each DDBMF process will receive periodically (n-1) messages from the other processes. Now assume that a DDBMF process receives a request for access to one of the files under its control from a local or remote user process. If the requesting process belongs to class 1, the DDBMF will respond immediately to the request. Otherwise the DDBMF will delay action until the next time interval, i.e., until receiving updated information about the status of the network files from other DDBMF processes. The request is then checked against any possible interference (deadlock) and the user process is notified once a decision is made.

Requests which can not be acted upon until the next time interval are placed in a pre-test queue.

> At the beginning of a clock interval, each processor receives information from other processors including the contents of the file queues and the pre-test queue. The processor extracts the contents of the pre-test queues and combines them to construct a global pre-test queue which includes all the requests for file access received by all processors during the previous time interval. The file access requests on the global pre-test queue are tested for deadlock conditions and decisions are then made.
>
> To avoid deadlock situations caused by critical race conditions, the file access requests on the global pre-test

queue must be arranged in the same order in all processors... All processors must then follow a predefined routine in constructing the global pre-test queue. The resulting versions of the global pre-test queue will be identical in all processors at the beginning of every clock interval.

III.3 Some Comments about the Proposed Schemes

The authors state that their schemes will work if records, or other units serve as the identifiable unit of object data, rather than files, which were mentioned throughout the paper. When records are allocated individually, there will be more message traffic due to additional message requests for access to database objects. Nowhere in the paper is the problem of message congestion at the control node (when using the Centralized approach) discussed. With all requests for access to database objects being handled by the central DDBMF, there exists the possibility of a message bottleneck at the control node, which would degrade network performance due to slow response to the requests.

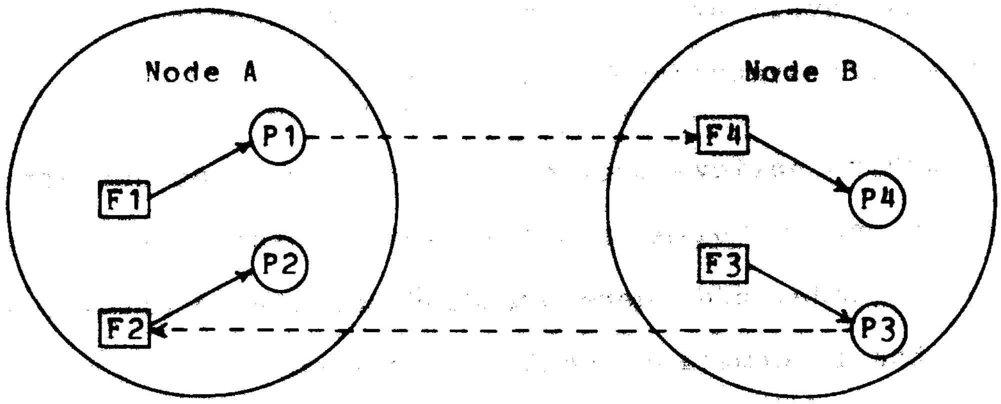
It is mentioned that failure of the control node (when using the Centralized approach) can "paralyze the operation of the whole system," although all the DDBMF's can send all their information to another DDBMF, thus recreating the global picture of the system at a newly designated control node. Although the author's Centralized approach may be "inefficient," it can be used to successfully detect all process deadlocks when only waits on database objects are involved.

The Decentralized approach, as described in the paper, does not detect all deadlock situations when only process waits for database objects are involved. Consider the following example:

Let the network consist of two nodes, A and B. Let processes P1 and P2, and files F1 and F2 be local to node A, and let processes P3 and P4, and files F3 and F4 be local to node B. Assume the following state of the network. (Figure III.1a contains a diagram for this "intermediate" state.) P1 has exclusive control of F1 and is queued waiting for access to F4, P2 is active (non-blocked) and has exclusive control of F2, P3 has exclusive control of F3 and is queued waiting for access to F2, and P4 is active and has exclusive control of F4. P1 and P3 belong to class 2, as defined by Mahmoud and Riordon, and P2 and P4 both belong to class 1 as long as each does not request access to a file located in a node other than the one in which the process resides.

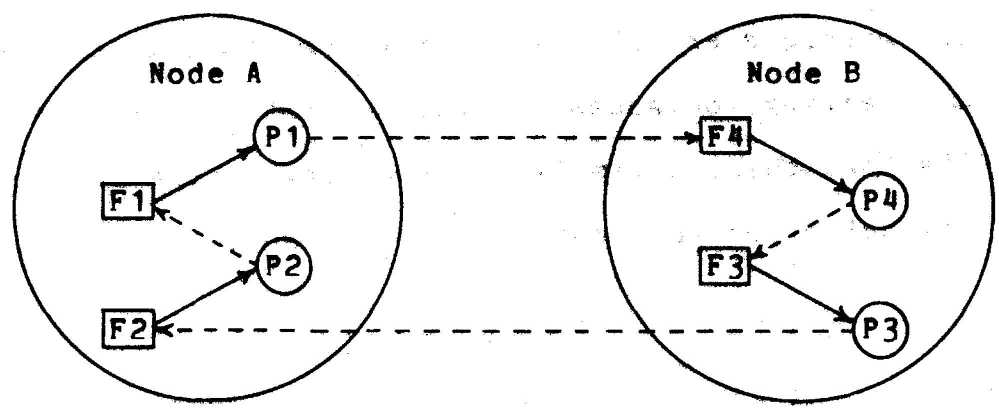
Now, within the same time interval, let P2 request access to F1 and let P4 request access to F3, thus creating a deadlock because neither file can become available. (Figure III.1b contains the final state diagram for this deadlock.) P2 and P4 remain class 1 processes, and therefore these requests should be acted upon immediately and each node will check for deadlock using the information that it has. No deadlock will be detected because neither node has the information about the recent request in the other node, and no provisions are stated in the article which imply that deadlock involving P2 or P4 will be checked for at the onset of the next synchronous clock period.

The authors believe that class 1 processes do not contribute to deadlocks that involve processes waiting for files located in more than one node, and therefore deadlock can be checked for using only the information located at one node when a class 1 process requests access to a file. It is this assumption that leads to the downfall of their Decentralized approach, because it is possible that a class 1 process will request access to a file controlled by a class 2 process, resulting in a deadlock (as shown in the previous example) involving processes which are collectively waiting for access to files located in two or more nodes. Note that this is similar to the flaw in the protocols for deadlock detection proposed by Chandra, Howe and Karp.



Intermediate State Diagram

Figure III.1a



Final State Diagram

Figure III.1b

KEY

Represents a process

Represents a file

Represents a process having exclusive use of a file

Represents a process waiting for access to a file

IV. Introduction to Proposed Solutions

The deadlock detection schemes that are presented in Chapters V and VI are based on the creation and expansion of ordered blocked process lists (OBPL's) and the restriction that a process may only have one unapproved outstanding resource request (and therefore be waiting for at most one resource at any instant). A resource may be any non-ambiguously defined portion of an object, whole object, or collection of objects which are requested as an entity and released as an entity by all users. (The case where there are several equivalent resources like tape drives is not considered. A discussion of physical devices occurs later in this chapter.) An OBPL is a list of process names, each of which (with the exception of the last process in the list) is waiting for access to a resource that has been assigned to the next process in the list. Each process name in the list is often referred to as a process entry in the OBPL, and when an OBPL is sent between nodes, a resource name is inserted into the single resource identification portion of the OBPL. The last process to have an entry in the OBPL is either waiting for access to the resource named in the resource identification portion, or it already has access to that resource. In the former case, it must be determined what process controls the resource, whereas in the latter case, the state of the last process in the OBPL must be determined.

It is assumed that at each node there is a process management module (PMM) which will handle deadlock detection and resource allocation. It will maintain local state tables which will contain information about local resources (resources which are located in that node) and local processes (processes which are running in that node). If a PMM is checking for deadlock, and it is examining the OBPL with process entries P1, P2, ..., PN, then it knows that each process in the list (with the exception of PN) is waiting for the next process in the list to release a desired resource. If PN is not blocked, there is no deadlock and the OBPL can be discarded. If it is blocked, then a PMM must find out what process has been allocated the resource for which PN is waiting. If this process already has an entry in the OBPL, there is a deadlock, otherwise a PMM must append the process name to the OBPL and repeat the above. The schemes that are being proposed differ from each other in the way the OBPL's get expanded.

IV.1 Descriptions of Resources

There are three types of resources that a process may wait for where the blocking of the process can result in a deadlock. They are database objects, message text from other computerized processes, and message text from operators (manual processes). A distinction is made between message text from processes and message text from operators because a deadlock which involves no operator messages can be detected without operator interaction, whereas if a process is waiting for message text from an operator, a deadlock can not be detected without the operator stating what he/she is waiting for. The reason for the latter point is that an operator typically does not type in "receive message" statements, but accepts output as it is given. In the algorithms presented, it is assumed that an operator can only wait for a message from a process with which he/she is communicating (a discussion of operator and process communication is given later in this section). This restriction can be relaxed, and it is discussed in Chapter VIII.

Database objects, as discussed in this paper, can be fields, records, files, or any other logical or physical component of a database. It is important that all processes treat the same portion of a database identically for the purposes of allocation. The level of granularity (which may vary for different database objects) at which database objects are allocated is unimportant for the detection of deadlock; it does however, affect the frequency of deadlock and, conversely, the burden of maintaining information about resource allocation.

Message text must be treated differently from database objects because once a message text has been assigned to a process, it is not available to any other process. In this sense, once a message text has been assigned, it no longer exists for future assignment. To ensure that a process receives the proper message text, the sending and receiving processes must create a unique connection over which message text between the two processes may pass. When a process would like to receive message text, it must state over which previously established connection the text should come. Similarly, when a process wants to send message text, it must give the message text and name the connection over which the text should pass. All messages that are sent and received over a given connection will be referred to as text within a specific message group.

When message text is sent by a process, it is queued for receipt at the proper destination end of the connection. A process may send several items of message text over a given connection before any messages are requested by the other process associated with the connection. In this case the items of message text are queued for receipt in a first in, first out manner. It is assumed that message management has infinite queueing capacity, and therefore the possibility of a deadlock involving a process which wants to send a message but is blocked because there is no place to put the message text will not be dealt with.

Unlike process to process messages, which may be sent between nodes, when a process and an operator communicate, they must be located at the same node. Similarly, however, an "operator connection" must be established between the operator and process before message text can be sent over the connection. The operator connection must be specified when message text is sent or received over the connection. When messages are sent from a process to an operator, they are usually printed immediately at the operator's terminal. However, messages that are sent from an operator to a process are queued for receipt in the same manner as process to process messages.

All of the resources described above are uniquely identifiable, and are allocated dynamically (i.e. during the execution of the process requesting access to the resource). None of them are physical devices (tape drives, printers, etc.), which are often not uniquely identifiable (there may be N of a kind). Physical devices are not considered by the algorithms that are being proposed because they are typically allocated to a process before execution begins and the known networks restrict processes to requesting physical devices at the same node. (If a process wants to control a physical device at another node, it must do so indirectly through a process located at the same node as the desired device.) Additionally, transaction oriented processes typically do not use dedicated devices.

IV.2 Access to Resources and the Blocking of Processes

A process may get blocked when it requests read only (shared) access or exclusive (read/write) access to a database object. While one process has exclusive access to a specific database object, all requests for access to that database object result in the requesting process being blocked. While at least one process has shared access to a specific database object, all requests for exclusive access to that database object result in the requesting process being blocked, and requests for shared access to that database object will result in the requesting process being blocked or being granted access to the desired resource (depending upon the resource allocation scheme in use).

Because data values are not changed when a process only reads a database object, any number of processes may be allowed to have concurrent read only access to a database object. When all processes that had shared access to a given database object have released it, or when a process releases a database object from exclusive use, at least one process will be awakened and granted access to the newly released database object, if any were waiting for access to it.

Once a process has been granted shared access to a specific database object, subsequent requests by that process for exclusive access to that database object are rejected. This restriction prevents a process from getting blocked waiting for a database object that it already has access to, and implies that a process must declare its most restrictive use when it requests access to the database object. (It must request exclusive access if there is any chance that the process might change the content of the database object.) In order to ensure that a process has a consistent view of the database, and that processes may be rolled back to a previous state (when necessary), no database objects will be released by a process until that process has reached a "commitment point", at which time all the database objects that the process had access to are released. A commitment point is always reached at process termination. (When a process continues processing after reaching a commitment point, for purposes of detecting deadlock, a PMM can treat it as a new process because it released all its database resources, and notified all processes to which it could send messages that no more messages are forthcoming. The external effects of a process, including database updates and message text sent, can not be cancelled after commitment. Process commitment points are synchronized, which is to say that after a process reaches a commitment point, it does no further processing until all processes with which it has established connections over which it can receive messages have also reached commitment points.)

If a process attempts to receive message text over a specific connection, it will be given one message if any are queued for receipt at that process'es end of the connection. If no messages are available, the process is blocked until message text arrives. Upon arrival of a message, the process will be awakened, because the receiving process is uniquely identified by the connection over which message text is sent. Steps must be taken to ensure that the receiving process and the sending process of a message treat the same text as one message. (One process can't treat a line as a message when the other process treats a group of sentences as a message.)

IV.3 Creation and Expansion of an OBPL

When a PMM wants to check whether a given blocked process is involved in a deadlock, it creates an OBPL and inserts the network unique name of the process as the first process entry in the OBPL. (It is assumed that operators, processes and resources have unique names within a node, and these names can be made unique within a network by qualifying them with the name of the node in which they reside. Throughout this Thesis, operator, process and resource names are assumed to be network unique.) Call this process P1. Let R1 be the resource to which P1 desires access. R1 is then inserted into the resource identification portion of the OBPL. A PMM (which PMM depends upon what scheme is being used to detect deadlock, and whether P1 and R1 are in the same node) then determines what process controls R1. If R1 is a database object, then the process that controls R1 is the process that has access to it. (If there are several shared readers of R1, then it is said that each reader controls R1 and the OBPL is copied enough times so that there is one list for each reader of R1, and a different copy of the OBPL is used for each reader.) If R1 is message text in a message group, then the process that controls R1 is the process that can send the desired message, and if R1 is message text from an operator connection, the process that controls the resource is the human operator that can send the message. If R1 is message text over a connection to which no process other than P1 has associated itself, the PMM saves the OBPL so that after another process or operator associates itself with the connection the needed information will be available and the OBPL can be expanded further. It is assumed that no deadlock can exist unless two processes are associated with the connection over which the desired message text can be received.

Let PK be the process that controls R1. A PMM then checks if PK already has an entry in the OBPL that is being examined. If it doesn't, the PMM adds its name to the OBPL and then lets some PMM determine if PK is active. If PK had an entry in the OBPL, the PMM has detected a deadlock, and should take the appropriate action. Note that the entry for PK can be anywhere in the OBPL, as it is possible that a process not involved in the deadlock may be waiting to access a resource controlled by a process that is involved in the deadlock. If PK is active, then there is no deadlock and the OBPL can be discarded. If PK is blocked, then the above procedure should be repeated, except PK should be used instead of P1 and a PMM determines what resource PK is waiting for. If PK represents an operator, then the PMM must save the OBPL until information about the status of the operator becomes available. A message is sent to the operator stating that this state information is desired. If the operator sends message text to a process, or if the operator responds that he/she is active, then all OBPL's that needed state information about this operator are discarded since there is currently no deadlock. If the operator states that he/she is waiting, then the operator connection over which the operator is awaiting a message must also be stated. The process that can send the operator the desired message is determined from the connection name, thus the PMM now knows what process controls the resource the operator desires, and this information is used to further expand all the OBPL's that needed state information about the operator. If no OBPL's needed this information, and the operator volunteers the information that he/she is blocked, then an OBPL is created with the first process entry representing the operator.

In order to ensure that a PMM sees a consistent set of state tables, no resources get allocated or released in the node of the PMM while the PMM is examining an OBPL. (The PMM holds exclusive use of the state tables in its node. The reason for this restriction becomes apparent in Chapter VI in the verification of the decentralized algorithm.) There is no chance of a PMM itself being involved in a deadlock because it is the only process that has access to the state tables in its node, and it does not wait for any messages or request access to any other database objects. Resource requests and OBPL's arriving from other nodes result in subroutine calls to the PMM. These calls are handled in a FIFO sequence. In addition, when a process or operator associates itself with a connection, a PMM is called to check if any OBPL's have been saved waiting for this information. Furthermore, when an operator sends message text to a process or states that he/she is active or blocked, the PMM at that node checks if any OBPL's have been saved waiting for state information about the operator and takes the appropriate action.

The time at which an OBPL gets created depends upon the optimization of the deadlock detection scheme, and which PMM creates the OBPL depends upon what scheme (centralized/decentralized) is used. An OBPL can be created as soon as a process becomes blocked, or it can get created after

'X' units of time have elapsed without the process gaining access to the desired resource. The latter approach will be used with the expectation that normally the process will be granted access to the desired resource within 'X' units of time because deadlock does not exist. Thus the overhead involved in creating and expanding an OBPL will usually be avoided. However, within the body of this paper, in the interest of clarity it is assumed that an OBPL is created immediately after it is determined that a desired resource is currently unavailable. It should be understood that the removal of this assumption, and the imposition of a delay before the OBPL gets created, does not impair the effectiveness of the algorithms because once a deadlock occurs, it exists until some type of recovery action is initiated.

Certain information must be available to the PMM's if the OBPL's are to be properly expanded. The PMM at each node will maintain a table which has an entry for each process in its node. Associated with each process entry will be a list of all the resources to which the process currently has access, and the name of the resource to which the process desires access (if the process is waiting). For each resource at the node, the PMM must keep information stating what process or processes currently have access to that resource, and what type of access they have. In addition, a list of all processes that are waiting for access to that resource must be maintained. (The latter information is necessary so that the resources will be properly allocated when they become available.)

V. Centralized Approach to Deadlock Detection

A "centralized" approach to deadlock detection in a computer network is based upon the premise that one node (the "control" node) in the network will act as the center of activity for global resource allocation and deadlock detection. In order to reduce overhead, any requests for resources or checks for deadlock that can be handled entirely by one node should not request the service of the control node. For reasons that will be explained later, the following description has not been refined, and should not be viewed as a working algorithm. The description presents some ideas that could form the basis for a practical centralized approach to deadlock detection.

V.1 Allocation of Resources

A process management module (PMM) will have responsibility for granting access to a local resource as long as no remote processes have been allocated the resource nor have been queued for it. When these conditions do not hold, the control process management module (CPMM) (located in the control node) will have responsibility for granting access to the resource. Thus when a process desires a remote resource, the request must go to the CPMM. When a process requests a local resource, the request must go through the CPMM only if that module currently has responsibility for granting access to the resource, otherwise the request will be handled by the local PMM. The set of resources for which the CPMM grants access changes dynamically. (As soon as a process requests a remote resource, that resource becomes a member of the centrally managed set if it isn't already a member, and when the conditions above are satisfied again, the resource is removed from the set.) For each resource in the set, the CPMM maintains a list (in the global resource control table) of all processes queued for that resource plus the name of the process or processes (in the case of shared access) that have been allocated the resource.

There are essentially three classes of resource requests in this type of network. The following is a list of the resource request classes and the proper response to each type of request:

1. A process requests a resource at the same node as the process, and the local PMM is responsible for granting access rights to the resource: The PMM can block the process or give it the resource. In either case, the PMM can update the appropriate tables.

2. A process requests a resource at the same node, and the CPMM has been given responsibility for granting access rights to the resource: A message containing the resource request must be sent from the local PMM to the CPMM. The local PMM will block the process until it receives notification from the CPMM that the desired access has been granted. Upon receipt of the resource request, the CPMM will either grant the process access to the desired resource, or keep it blocked. In either case, the CPMM updates its tables to reflect the state after this request has been processed.

3. A process requests a resource at another node: A message containing the resource request must be sent from the local PMM to the CPMM. The local PMM will block the process until it receives notification from the CPMM that the desired access has been granted. Upon receipt of the resource request, the CPMM, if it had the responsibility for granting access to the specified resource, will either grant the process access to the desired resource or keep it blocked. If the CPMM did not have such responsibility, it will demand it from the PMM that does, and then the CPMM will process the request. After the request has been processed, the CPMM

will update its tables appropriately.

When a process reaches a commitment point, the local PMM will release all the resources that the process controlled. The PMM can then grant other local processes access to the resources that were released and for which it has responsibility for granting access. If any resources which were under the CPMM's control were released, the CPMM will be notified of the reaching of a commitment point by the process, and it will then grant other processes access to the resources if any are queued for them and the rules for resource allocation permit the new assignments. If possible, following a resource release, the CPMM will return responsibility for granting access to a resource back to the PMM in the node where the resource resides.

V.2 Deadlock Detection

When a PMM denies a request for a resource and blocks a process, it then creates an OBPL with a process entry for the blocked process. It then expands the OBPL until 1) a deadlock is detected, 2) it is ascertained that there is no deadlock, or 3) the PMM does not have enough information to expand the OBPL further (because an involved process is waiting for a global resource, or a local resource is controlled by a remote process). In the latter case the PMM sends the OBPL to the CPMM, which will complete the expansion of the OBPL. When the CPMM denies a request for access to a resource, it creates an OBPL with a process entry for the blocked process and then expands the OBPL until a deadlock is detected or it is ascertained that no deadlock exists.

To expand an OBPL, a PMM uses its state tables that were described in Chapter IV, and the CPMM uses its global resource tables and those of the PMM's in the network. (How it obtains copies of these tables is discussed later in this chapter.) The method by which the PMM's expand an OBPL will be described first, and it will be followed by the method which is used by the CPMM. After a PMM has created an OBPL, it acts as if it were in step 2 below, with PN set to the name of the process which was just blocked, and RN set to the name of the resource for which PN is waiting. The following is a list of steps taken by a PMM when expanding an OBPL:

1. Let PN be waiting for resource RN. If RN is a local resource, go to step 2, otherwise go to step 6.

2. If RN is controlled only by local processes, go to step 3, otherwise go to step 6.

3. Let PX be the process controlling RN. If PX is blocked, go to step 4, otherwise there is no deadlock and the OBPL can be discarded. (If there are J shared readers of RN, repeat this step once for each reader.)

4. If PX is already contained as a process entry in the OBPL, there is a deadlock and the PMM must take appropriate action. If PX is not in the OBPL then go to step 5.

5. Append PX as a process entry in the OBPL and go to step 1, where PX is used in place of PN.

6. Place RN into the resource identification portion of the OBPL and send the OBPL to the CPMM. Halt.

The CPMM will create an OBPL when it denies a request for access to a resource. The only process entry in the newly created OBPL is for the process whose resource request could not currently be honored. After a CPMM has created an OBPL, it starts in step 1 below, with RN set to the resource whose unavailability resulted in the OBPL being created. If the CPMM receives an OBPL from a PMM, it sets RN to the resource that was placed in the resource location of the OBPL, and sets PN to the last process to be inserted into the OBPL. The CPMM verifies that PN is still waiting for RN (if it isn't, either RN has already been allocated to PN or the CPMM has not yet received the request by PN for access to RN, so there is currently no deadlock and the OBPL can be discarded) and then starts in step 1 below. The following is a list of steps taken by the CPMM when expanding an OBPL:

1. Let PX be the process controlling RN. (If there are J shared readers of RN then repeat this step once for each reader.) To find PX, the CPMM first checks if RN is in the global resource table. If it is, then this table is used to get PX, otherwise the copies of the local tables for the node in which RN resides are used by the CPMM. Go to step 2.

2. If PX is blocked, go to step 3, otherwise there is no deadlock and the OBPL can be discarded. (First check if PX is waiting for a global resource, and if it isn't, then check the copies of the local tables for the node in which PX resides in order to find out if PX is blocked or active.)

3. If PX is already contained as a process entry in the OBPL there is a deadlock and the CPMM must take appropriate action. If PX is not contained in the OBPL, go to step 4.

4. Append PX as a process entry in the OBPL and go to step 5, where PX is used in place of PN.

5. Let PN be waiting for RN. (If PN is waiting for a global resource, use the global resource table to determine RN, otherwise use the copy of the local tables for the

node in which PN resides.) Go to step 1.

V.3 Issues to be Resolved

There are several problems with the algorithm as described in the previous section. A major problem is determining how the CPMM maintains its copies of the tables belonging to the PMM's in the network. One possibility is to have each PMM send a copy of its tables to the CPMM every 'X' units of time. Another is to have the CPMM request a new copy of the tables that it needs if 'Y' units of time (Y may equal 0) have elapsed since it last received a copy of the desired table. In either case, once a deadlock has been detected, all the tables of the nodes whose processes and resources are involved should again be requested by the CPMM in order to verify that the deadlock exists and that the CPMM's detection was not a result of the CPMM looking at an inconsistent state of the network. (Due to the fact that the list of resources that are kept in the global resource table changes dynamically, and the CPMM does not always have an up to date copy of the local tables, it is possible that some needed information may be incorrect and could cause problems for the CPMM.) It is probable that there are better and more reliable methods of maintaining the copies of the local tables in the CPMM.

When the CPMM is expanding an OBPL, and encounters a process waiting for message text from an operator, it can be difficult to get the needed state information. A method is needed whereby the

CPMM can save the OBPL and notify the PMM at the node in which the operator resides, that this state information is desired. The PMM must then query the operator and send the CPMM this information along with its latest state tables.

Another problem that must be resolved occurs when related messages cross between two nodes. An example of this is that the CPMM may return the rights to grant access to a resource to a PMM at the same time that the PMM under discussion sends a request to the CPMM stating that one of its processes would like to access that local resource. Care must be taken when designing the resource allocation scheme to ensure that cases like this will be detected and the desired action (which in this case is granting the process access to the resource) will occur. In addition, steps must be taken in the deadlock detection algorithm to account for and detect similar problems.

V.4 Reasons for not Refining the Algorithm

Several factors led to the decision not to refine the above algorithm to the point where it could easily be proved to work. It was felt that with all remote resource requests going to one node, there would be message congestion at that node, plus there would be an extra delay due to the fact that a request must go through the central node rather than going directly to the node in which the desired resource resides. Another factor that influences message congestion is the size of the tables that will get sent from the PMM's to the CPMM. Since database records may be considered resources, these tables can get quite large, and it would be preferable to only send the CPMM parts of these tables, but then there is the problem of deciding which parts should be sent, and what the CPMM should do when it was not sent enough information.

When one node is used as the center of activity in a network, the network becomes only as reliable as that node. It would be possible to have another node in the network serve as a backup to the CPMM and maintain copies of the CPMM'S tables. There would be a delay in updating this duplicate copy, and it would have to be decided how often the copy should be updated. (A great deal of overhead is involved if a message is sent to the "backup" node every time the CPMM changed its tables.) It would also be possible to reconstruct the CPMM's tables at another node by requesting information from all other nodes in the network, thus saving the overhead involved in maintaining the duplicate copy at a cost of added delay if the control node were to become inoperable for some reason. In a computer network it is desirable to distribute the computing and to minimize the overall network problems when one node crashes. This was the major reason it was decided not to spend time refining an algorithm for deadlock detection which relies upon one node in the network.

VI. Decentralized Approach to Deadlock Detection

A "decentralized" approach to deadlock detection in a computer network is based upon the premise that there should be no central or control node and that all nodes in the network will share the responsibility for detecting deadlocks. In addition, the failure of one node should only affect the processes of that node and the processes of other nodes which are accessing that node's resources. The amount of duplicate process and resource state information among the various nodes in the network will be kept to a minimum, and each node will be requested to help check for a deadlock only when at least one of its processes or resources is involved.

VI.1 Allocation of Resources

A process management module (PMM) located at each node will always have responsibility for granting access to resources located at that node. Whenever a process requests a resource, the request will be processed by the PMM at the same node as the process. This PMM will determine if the desired resource is local or if it is located at a different node. (Message text should be treated as local to the node of the sending process.) If it is a local resource, then the PMM can immediately determine if the desired access may be granted or if the process must be blocked waiting for the availability of the resource. If the request is for a remote database object, then the PMM must block the process and send a remote database object request (RDOR) to the PMM in the node which contains the desired resource. Upon receipt of an RDOR from another node, a PMM will determine if the requesting process must remain blocked or if it may be granted access to the desired resource. If access is granted, a remote database object assignment (RDOA) is sent to the PMM in the node in which the requesting process resides. Upon receipt of this RDOA, the PMM will awaken the proper process and notify it of the resource assignment. If the process must remain blocked, no message is sent to the node in which the process resides. The details of implementing this feature are not described, as they are not relevant to the scope of this Thesis.

When a process reaches a commitment point, the PMM at its node will release all the database resources that the process had access to and notify the necessary processes that no more messages are forthcoming from the specified process. All local resources can be immediately allocated to other processes in accordance with the rules for resource allocation, and messages must be sent to all nodes which had resources allocated to the process, informing their PMM's of the reaching of a commitment point. Upon receipt of such a message, the PMM will appropriately update its tables and assign the resources to other processes in accordance with the rules for resource allocation.

VI.2 Deadlock Detection

When a PMM determines that a resource at its node can not currently be allocated to a process that requested it, the PMM creates an OBPL (ordered blocked process list) with a process entry for the blocked process. It then expands the OBPL until 1) a deadlock is detected, 2) it is ascertained that there is no deadlock, or 3) the PMM does not have enough information to further expand the OBPL. (Note that if a database object has been requested, the OBPL is created in the node where the database object resides, whereas if message text has been requested, the OBPL is created in the node where the requesting process resides.) The PMM starts expanding the newly created OBPL in step 10 below. When a PMM receives an OBPL from another node, it starts in step 1 below in an attempt to complete the expansion of the OBPL. The reasoning behind each step is contained in the next section, and these explanations should be read before one attempts to verify the correctness of the algorithm. It should be noted that within the algorithm, PX and RX are names of variables whose contents represent processes and resources, respectively, even though they are sometimes used as though they were process and resource names themselves.

1. Set RX to the value contained in the resource identification portion of the OBPL. If RX represents a resource which is local to the node expanding the OBPL, then go to step 2, otherwise go to step 8.

2. Verify that the last process added to the OBPL is still waiting for RX. If it isn't then discard the OBPL and halt, otherwise go to step 3.

3. Let PX be the process controlling RX. (If there are J shared readers of RX, then repeat this step once for each reader.) If PX already has a process entry in the OBPL, then there is a deadlock and the PMM must take the appropriate action. If PX is not in the OBPL then go to step 4.

4. If PX represents a process which is local to the node expanding the OBPL, then go to step 5, otherwise go to step 7.

5. If PX is active, there is no deadlock, so discard the OBPL and halt. Otherwise go to step 6.

6. Append PX as a process entry in the OBPL and go to step 10.

7. Append PX as a process entry in the OBPL. Place RX into the resource identification portion of the OBPL and send the OBPL to the PMM in the node in which PX resides. Halt.

8. Verify that the last process added to the OBPL still has access to RX. If it doesn't, discard the OBPL and halt. Otherwise go to step 9.

9. If the last process added to the OBPL is active, there is no deadlock, so discard the OBPL and halt. Otherwise go to step 10.

10. Get the name of the resource for which the last process added to the OBPL is waiting and call it RX. If RX represents a resource which is local to the node expanding the OBPL, go to step 3, otherwise go to step 11.

11. Place RX into the resource identification portion of the OBPL and send the OBPL to the PMM in the node in which RX resides. Halt.

VI.3 Explanation of Steps in the Deadlock Detection Algorithm

The following is a description of the reasons for including each step in the deadlock detection algorithm described in the previous section. Each numbered paragraph below corresponds to the step with the same number in the previous section.

1. An OBPL will be sent to a node when it must be determined what process controls a given resource, or what state (active or blocked) a given process is in. If the resource that was named in the resource identification portion of the OBPL is local to the node that just received the OBPL, then in order to expand the OBPL the PMM needs to know what process has access to that resource and it goes to step 2, otherwise it goes to

step 8 in order to check the state of the last process to be added to the OBPL.

2. It must be verified that the last process added to the OBPL is still waiting for RX because it is possible that while the OBPL was sent from the PMM in the node containing the process, the PMM in the node containing RX sent a message stating that the process has been granted access to RX. If this process is no longer waiting for RX, the state that was assumed when the OBPL was sent no longer exists, and the OBPL can be discarded.

   If RX represents a database object, then the last process added to the OBPL is still waiting for RX if it is still queued for access to the database object. If RX represents a message in a message group, then RX is qualified by the sequence number of the message within the message group that is desired. (If the process has already received N messages over the specified connection, then it is waiting for message number N+1 in the message group.) The process is still waiting for the specified message only if the number of messages already sent to it over the given connection is less than the number that qualified the message group name.

3. If PX already has a process entry in the OBPL, then there is a loop of processes each waiting for a resource that is controlled by the next process in the loop, so a deadlock has been detected. If PX does not have a process entry in the OBPL, go to step 4 in order to expand the OBPL further if PX is not active.

   If RX is a database object which has J shared readers, then a copy of the OBPL must be made for each of these readers because the process that requested access to RX will not be able to access RX if the process is in a deadly embrace loop involving any one of the J readers.

4. If PX is local to the node which is expanding the OBPL, then the PMM can immediately check the state of PX, so it goes to step 5. If PX is not a local process, the OBPL must be sent to the node in which PX resides, so the PMM goes to step 7.

5. If PX is not currently blocked waiting for access to any resources, there can be no deadlock currently involving PX. If PX represents an operator, the OBPL must be queued waiting for state information about the operator. The PMM will then ask the operator to enter information about his/her state. The acceptable operator responses are 1) that he/she is waiting for a message over a given operator connection, 2) that he/she is active, or 3) a

regular message over an operator connection. If the operator sends a regular message, or states that he/she is active, then there is no deadlock and all the OBPL's that are queued for state information about this operator will be discarded. If the operator states that he/she is waiting for a message, then the PMM can (by the use of the given operator connection) determine what process can send the message that the operator desires, and the PMM can then further expand the OBPL. It may be desirable to "time out" a non-responsive operator, as operator inaction can stall the system and perpetuate an undetected deadlock.

6. PX is blocked, so insert it as the last entry in the OBPL and then go to step 10 in order to further expand the OBPL.

7. Insert PX as the last entry in the OBPL even though the PMM does not know the state (active or blocked) of PX. (This will be checked by the node that will receive the OBPL.) Place RX into the resource identification portion of the OBPL to indicate that PX currently controls RX, and the state of PX is needed information. If RX represents a message within a message group, it is qualified by the sequence number of the message within the message group that is desired. The PMM therefore sends the OBPL for further expansion to the PMM in the node which contains PX.

8. It must be verified that the last process added to the OBPL still has access to RX because it is possible that while the OBPL was sent from the PMM in the node containing RX, the PMM in the node containing the process sent a message stating that RX has been released by the process. If the process no longer has access to RX then the state that was assumed when the OBPL was sent no longer exists, and the OBPL can be discarded.

9. If the last process added to the OBPL is not currently blocked waiting for access to any resources, there can be no deadlock currently involving the process. If the process is blocked, the PMM goes to step 10 because the process already has been inserted as the last process entry in the OBPL.

10. Step 10 can be reached from step 6 or step 9. In either case, the last process added to the OBPL is local to the node which is expanding the OBPL, so the PMM can find out what resource the process desires access to. Set RX to the name of this resource. If RX is local to the node that is currently expanding the OBPL, the PMM can continue to expand the OBPL, so it goes to step 3,

otherwise it goes to step 11.

11. To further expand the OBPL, what process has access to RX must be known, so the PMM sends the OBPL to the PMM in the node in which RX resides. Place RX into the resource identification portion of the OBPL to indicate that the last process added to the OBPL is blocked waiting for access to RX and what process controls RX is needed information. In the case where RX represents a message within a message group, it is qualified by the sequence number of the message within the message group that is desired. Send the OBPL for further expansion to the node in which RX resides.

VI.4 Verification of the Algorithm

There are two parts in the verification of the correctness of the decentralized algorithm for deadlock detection. The first and most important part is to prove that all deadlocks get detected. The second part is proving that a deadlock is not "detected" when (except in a special case discussed later) one does not exist.

Part 1

To prove that all deadlocks get detected, it will be shown that once a deadlock state is reached, an OBPL will be created that will be passed among nodes which will expand it until the deadlock is detected. There are two assumptions that are required for this proof: 1) All internodal messages eventually get received by the proper nodes (and therefore no OBPL's are "lost" in the transmission between nodes), and 2) while the OBPL is being expanded, none of the processes involved in the deadlock are aborted (which would break the deadlock before it is detected) or rolled back to a previous state (which would imply the deadlock has been detected by the expansion of another OBPL).

Let a deadlock consist of processes P1, P2, ..., PN, with P1 waiting for a resource controlled by P2, ..., and PN waiting for a resource controlled by P1. (Process names are unique within a node and they can be made network unique by qualifying them with their node names, so throughout this proof, assume the Pi represent distinct processes.) When each process, Pi, involved in the deadlock was denied access to a resource controlled by another process in the deadlock, an OBPL was created with the first process entry representing Pi. One of these OBPL's must have been the last (in time) to be created, thus the deadlock existed at that time. (If two or more of these OBPL's were created simultaneously and they were the last to be created for processes involved in the deadlock, then any one in this "last group" may be arbitrarily selected as the last to be created. The important point is that the deadlock existed at the time the OBPL was created, and all the relevant tables collectively contain the information showing each process in the deadlock waiting for a resource controlled by another process in the deadlock.) For simplicity, assume that this last OBPL contains P1 as its first process entry. Additionally, in the ensuing discussion, a message from an operator to a computerized process will not be treated as a special type of resource because it is assumed that operators will state what they are waiting for when asked to do so by a PMM.

After P1 has been inserted as the first process entry in this "last" OBPL, the PMM which will begin the expansion of the OBPL will be in step 10 of the algorithm. If P1 is waiting for access to a resource local to a different node, then the PMM executes steps 10 and 11, and another PMM (after receipt of the OBPL) executes steps 1 and 2, then goes to step 3, otherwise the PMM executes step 10 and goes to step 3. (Since there is a deadlock, the OBPL will not be discarded.) Now, no matter what P1 is waiting for, it can be assumed that a PMM is about to start step 3 and it can (i.e. it has the information in its tables) determine what process (in this case, P2) controls the resource P1 has requested. There are two ways (depending on whether P2 is local or global to the node in which the OBPL is currently located) in which a process entry for P2 will be inserted into the OBPL.

Case A: P2 is "local".
Steps 4, 5 and 6 are executed, then step 10 will be executed. The PMM will then be ready to execute step 3 or it will execute step 11 and another PMM will execute steps 1 and 2, and will be prepared to execute step 3.

Case B: P2 is "global".
Steps 4 and 7 are executed, then the PMM which then receives the OBPL will execute steps 1, 8, 9 and 10. It will then be ready to execute step 3 or it will execute step 11 and another PMM will execute steps 1 and 2, and will be prepared to execute step 3.

This "last" OBPL now has process entries for P1 and P2, and a PMM is about to execute step 3 to continue the expansion of the OBPL. A PMM is now essentially in the same position some PMM was in shortly after the OBPL was created. The only difference is that now two processes have entries in the OBPL, and RX is set to the resource for which P2 is waiting, rather than the resource for which P1 is waiting. By repeating the above procedure as many times as necessary, the OBPL will be expanded to include process entries for processes P1, P2, ..., PN. At this point, when step 3 is executed, it will be determined that P1 controls the resource PN has requested, and the deadlock will be detected.

QED Part 1.

Part 2

To prove that every deadlock that gets "detected" actually is a deadlock, it must be shown that an OBPL will be discarded whenever there is a change in the state that was assumed when a process entry was made in that OBPL. (The one exception, which is ignored in the ensuing discussion, is the case where the assumed state changes due to the aborting or rolling back of a process, rather than having the state change due to a waiting process being awakened and granted access to the resource for which it was waiting.) This condition is sufficient because if a deadlock is "detected" when expanding the OBPL containing (in order of insertion) process entries for P1, P2, ..., PM, PN, and there has been no change in the state that was assumed when each process was entered into the OBPL, then P1 is still waiting to access a resource controlled by P2, ..., PM is still waiting to access a resource controlled by PN, and PN is still waiting to access a resource controlled by PJ, where PJ appears earlier in the OBPL. Thus a deadlock actually exists if one is "detected" and there has been no change in the state that was assumed when the process entries were inserted into the OBPL.

Assume that a PMM is expanding an OBPL with process entries (in order of insertion) P1, P2, ..., PK, PL. If the algorithm is correct, then P1 is waiting for access to a resource controlled by P2, ..., and PK waiting for access to a resource controlled by PL. Now assume that this state does not hold. That is to say, for some Pi, Pj with adjacent process entries in the OBPL, either Pi is not waiting for access to the same resource (say RQ) for which it was waiting when it was ascertained that Pi was blocked and that Pi should have an entry in the OBPL, or Pj no longer controls RQ. It will be shown that whenever this situation occurs, it will be detected and the OBPL will be discarded.

It can be assumed that Pi and Pj are PK and PL respectively, because if the state has changed from what was assumed when Pi was inserted into the OBPL, then it either changed before a PMM checked to see what Pj was waiting for, Pj was not blocked, or the state changed after there was a similar state change involving Pj and the next process in the list. (The latter claim can be made because if Pi was waiting for access to RQ which was controlled by Pj, and Pj controlled RQ and was blocked at the time that it was decided to further expand the OBPL, the only way the assumed state could change would be for Pj to incur a state change and be awakened so that it could release RQ.)

In order to show that PK is still waiting for RQ, and that RQ is still controlled by PL whenever it is decided that another process should be added to the OBPL, two cases must be considered. 1) PL, PK and RQ are all located in the same node, and 2) PL, PK and RQ are located in two or three different nodes in the network.

Case 1.

Due to the restriction that operators can only communicate with processes, there are three possible combinations of the types (process or operator) of PL and PK. (The resource type of RQ is either unimportant or uniquely determined by PK and PL.)

Case A: PK and PL are both processes.
Once PK has been inserted into the OBPL, and the PMM in the node in which PK resides is expanding the OBPL, the PMM determines that PK is waiting for access to RQ and that PL controls RQ. It then inserts PL into the OBPL if PL is blocked and discards the OBPL if PL is active. Since the PMM has exclusive use of the state tables in its node, there is no way the assumed state will change until after the OBPL is discarded, sent to another node or queued waiting for state information about an operator (in which case the state can not change until after the operator states that he/she is active or sends a message to a process, both of

which result in the OBPL being discarded).

Case B: PK is an operator and PL is a process.
PK is not inserted into the OBPL until the operator states that he/she is waiting for a message over a given operator connection (RQ). The PMM in the node in which PK resides then determines that PL is the process that can send the desired message. If PL is blocked, it is inserted into the OBPL, otherwise the OBPL is discarded. Since the PMM has exclusive control of the state tables in its node, the assumed state can not change until after the OBPL is discarded, sent to another node, or queued waiting for state information about an operator.

Case C: PK is a process and PL is an operator.
PL is not inserted into the OBPL until the operator states that he/she is waiting for a message over a given operator connection. PK is still waiting for a message from PL because the OBPL would have been discarded if any message text had been received from the operator since the OBPL was queued waiting for state information about the operator. (Note that it is possible that the desired message may have been sent by the operator before the OBPL was queued, but it has not been given to PK because calls to the PMM are processed in a first in, first out fashion. In this case though, the OBPL will be discarded before any state message from the operator is processed, because the desired message text was sent before the operator state message.) The OBPL will then either be discarded or have another process entry added to it, because an operator can only wait for a message from a process located at the same node.

Case 2.

Whenever an OBPL is sent between nodes, it must be verified that the state that was assumed when the OBPL was sent is still valid. Operators do not cause any OBPL's to be sent between nodes (because they only communicate with processes at their own nodes), thus in this discussion PK and PL are always processes. There are four combinations of the resource type of RQ and the locations of PK, PL and RQ.

Case A: RQ is a database object located in the same node as PK, but different from PL.
After it is ascertained that PK is blocked waiting for access to RQ, it is determined that PL controls RQ. PL is then inserted into the OBPL (after the entry for PK) and the OBPL is sent to the PMM in the node in which PL resides. When the PMM receives the OBPL, it first verifies that PL still controls RQ. If it doesn't, there has been a change in the assumed state (PL has released RQ), and the OBPL is discarded. Note that the OBPL is also discarded if it is determined that PL is not blocked.

Case B: RQ is a database object located in the same node as PL, but different from PK.
After it is ascertained that PK is blocked waiting for access to RQ, the OBPL is sent to the PMM in the node in which RQ and PL reside. Upon receipt of the OBPL, this PMM verifies that PK is still waiting for access to RQ. If it isn't, there has been a state change (PK was granted access to RQ), and the OBPL is discarded. The OBPL is also discarded if it is determined that PL (which controls RQ) is not blocked.

Case C: RQ is a database object located in a node which contains neither PK nor PL.
After it is ascertained that PK is blocked waiting for access to RQ, the OBPL is sent to the PMM in the node in which RQ resides. Upon receipt of the OBPL, this PMM verifies that PK is still waiting for access to RQ. If it isn't, there has been a state change, and the OBPL is discarded. If PK is still waiting for access to RQ, then the PMM inserts PL into the OBPL (since PL controls RQ) and sends the OBPL to the PMM in the node in which PL resides. After the OBPL is received, the PMM then checks that PL still controls RQ. If it doesn't, there has been a change in the assumed state, and the OBPL is discarded. The OBPL is also discarded if it is determined that PL is not blocked.

Case D: RQ represents message text and PK and PL are located in different nodes.
After PK is inserted into the OBPL because the process is waiting for message text in message group RQ, RQ is qualified by a message number.

The OBPL is then sent to the node in which PL resides. PL will only be inserted into the OBPL if it is blocked and the specified message has not been sent (which implies PK is still in the state it was in when it was inserted into the OBPL), otherwise the OBPL will be discarded.

It has been shown that whenever the relevant portions of the overall network state differ from the state that was assumed when process entries were inserted into the OBPL, the situation is detected and the OBPL is discarded. Therefore it is impossible to detect anything but deadlocks since a deadlock is never "detected" unless a PMM wants to insert a process into an OBPL when there is already a process entry in the OBPL for that process. It has thus been proven that the decentralized algorithm only "detects" deadlocks.

QED Part 2.

QED Decentralized Algorithm.

VI.5 Some Properties of the Algorithm

It should be noted that all references to processes in the previous sections actually referred to process "commitment units" (the period between commitment points), and the fact that commitment units within a process are network unique allows a deadlock to be detected at a node different from the one which contains the process that was found to already have a process entry in an OBPL. This situation can arise if the process under discussion controls a remote database object, and the PMM at the node in which the database object resides wants to insert the process into the OBPL due to its controlling the above mentioned database object. The OBPL need not be sent to the PMM in the node in which the process resides to verify that the process still controls the database object, because the process has not reached a commitment point (by virtue of the fact it already has an entry in the OBPL) and therefore has not released any database objects.

All resource requests will be handled with minimal delay because, for any request, the only nodes involved are those which contain the associated process and resource. (No information is needed from any other nodes to process the request.) The algorithm will function properly regardless of the resource allocation scheme in use, since the needed information about a resource is what process (or processes) currently controls it, not the order in which processes will be granted access to the resource in the future. (The latter information is necessary only for deadlock avoidance algorithms.)

While a PMM is expanding an OBPL, all other PMM's may be processing resource requests and releases. A PMM need only see a consistent state within its own node in order to expand an OBPL. The restriction that a PMM can not process resource requests and releases while it is expanding an OBPL can be removed if the decentralized algorithm is modified slightly. In step 10 the branch to step 3 would be eliminated (and therefore always go to step 11 after step 10), and then in step 11 a PMM may send an OBPL to itself. The new restriction would be that no resource requests or releases can be processed while a PMM is executing steps 1 through 11, although resource requests and releases could be processed between the execution of step 11 and step 1.

The same deadlock can be detected more than once if processes and resources located in two or more nodes are involved. This situation will occur if two or more processes request request resources at approximately the same time, resulting in OBPL's being created starting with different processes in the same deadlock loop. It is important to note that no matter how long it takes for OBPL's, remote resource requests, remote resource assignments, message text in message groups, and notification of a remote process termination to travel between nodes, the algorithm still functions as expected due to the verification steps that are included and the fact that once a deadlock exists, it will not be broken until after it is detected and recovery action is initiated.

VII. ADT Model of the Decentralized Algorithm

A functional model of the decentralized algorithm described in the previous chapter was designed and created using the facilities of the Architectural Definition Technique (ADT). The model was designed so that the algorithm could be easily tested. Additionally, by designing the model at the same time that the algorithm was being refined, several deficiencies of early versions of the algorithm were detected and corrected. (See section VII.2 and [1] for information about ADT.)

The model was written in PL/I and runs on the Honeywell Multics timesharing system. It was coded for ease of use and readability, and is not intended to suggest the most efficient way of implementing the algorithm in a computer network. A prerequisite to the use of ADT is an ability to understand the concept behind Data Structure Diagrams.

VII.1 Data Structure Diagrams

An information structure can be described by a Data Structure Diagram. A particular object in an information structure is referred to as an "entity", and an entire group of similar entities is called an "entity-class". (They are characterized by a prototype called an "entity-type".) The grouping that associates one or more entities of the same entity-class with one entity of a second entity-class (same or different type) in a subordinate relationship is known as an "entity-set". In a Data Structure Diagram, a block is used to represent an entity-type (the entity-type name is written inside the block). A "set-class" is a collection of similar entity-sets. (They are characterized by a prototype called a "set-type".) An arrow represents a set-type. It designates (by pointing from) the entity-type that "owns" the set-type and designates (by pointing to) the entity-type that serves as the "members" of the set.

There is a 1 to n relationship between the owner and members of an entity-set: n may be zero, one or more. For each owner there may be any number of members, but for each member, there is only one owner in any set occurrence. A dashed arrow is used to represent a set-type where the member relationship may or may not exist. This is called a "sometime" member relationship. When there can be only one member in an entity-set, a line (rather than arrow) is drawn between the owner entity-class and member entity-class. A dashed line is used when there can be a sometime one-to-one relationship.

A situation can arise where a set-type can have more than one type of entity in the member role. In this case a multihead arrow is used to represent the set-type. Similarly, a multitail arrow is used to represent a set-type where more than one type of entity can assume the owner role (although each member has only one owner). A more detailed explanation of Data Structure Diagrams can be found in [2].

VII.2 Architectural Definition Technique

ADT is an approach to arriving at a complete, concise, non-ambiguous functional specification of a software or hardware system which is totally independent of packaging considerations. To use ADT, one must describe the system state variables in terms of occurrences of entity-types, attribute types and set-types, and create a user interface as a set of machine processable function definition algorithms.

An example of an entity-type is "node" in a computer network. Each node in the network must have a name, which is an attribute of the entity. The entity-type and its attributes must be declared. In addition, all entity-sets which a node may belong to as a member or owner must be declared, and the relationship ("member", "owner", or "recursive") must be stated. A node is a member of the set of all nodes in a network, but it is the owner of various resources and processes located at that node. The manner in which entities and their attributes and set relationships are represented in the machine is irrelevant to the goal of achieving a functional specification. Therefore the ADT user is relieved of this burden.

A function definition algorithm is a body of code which specifies what action should take place in response to a given external stimulus. A function definition algorithm has several responsibilities. 1) It must validate the input parameters, 2) It must execute the logic of the function, 3) It must access the system state tables and update them appropriately to reflect the action taken, and 4) It must provide an external response representing the action (or lack thereof) that has taken place. A function definition algorithm usually includes a series of calls to the ADT modelling subroutines.

One integral part of ADT is a set of procedures which facilitate the modelling of the "system state". These procedures provide the capability to create and maintain a network structured database which holds the entities, attributes and relationships used to model the system.

A functional model created using ADT can be exercised and "validated" by the creation and execution of a sequence of commands. (Calls to the various function definition algorithms.) Any number of commands can be executed so that the model can be observed in order to determine if it acts in accordance with expectations.

Facilities are furnished in ADT to save these sequences of commands (scenarios) and to automatically execute them. There are also facilities so that the system state can be saved and restored. Display facilities are provided which permit a detailed examination of the system state without altering it. Using these facilities it is easy to construct experiments, alter them and examine the results at any time.

ADT is a deterministic system, and the machine is always in a stable state during the period between calls to the various function definition algorithms.

VII.3 The Deadlock Detection Model

The deadlock detection model which runs using ADT was designed to be driven entirely by the user of the model. All the nodes in the network must be created by the model user, as are processes and database resources located at each node. In addition all operators at each node must be declared. Each node in the network must have a unique name. Operator names and process names appear together in the same name space and must be unique within each node. They are qualified by the node name to make them unique in the network. Database objects must also have unique names within the set of database objects at a node.

Process wait situations may arise as a result of requests for message text in a message group or over an operator connection, or requests for access to a database object, but operator wait situations are not forced by the system because operators do not request message text, they only take it as it comes over an operator connection. All requests by processes for resources must be entered by the model user. The model will process the requests, and allocate the desired resources, if possible, otherwise the requesting process will be blocked. When message text is requested, the message group name (in the case of process to process communication) or operator connection name (for operator to process communication) must be given. With the model, before message text in a message group can be received by a process, the message group must first be initiated by the process which can send the messages, and then be accepted by the process that will receive the messages in the message group. (The model user specifies when this takes place.) Actual systems may allow message groups to be accepted by a process before another process initiates it. An operator connection must be established (by the model user) between an operator and a process at the same node before a process can receive message text over the operator connection. This model does not support the sending of messages from a process to an operator over an operator connection because typically messages from a process to an operator are not queued for receipt by an operator, they are simply printed at the operator's terminal without an explicit operator request.

In order to make the model easier to use, it was decided to make message group names and operator connection names unique within the network.

In a computer network it is probable that message text may be sent by either process involved in a connection through which they are communicating. (This is a two-way connection.) The model only allows the initiator of a message group to send message text over the associated connection because a two-way connection can be simulated using two one-way connections, with each process involved being the initiator of one of the message groups. The sender and receiver of message text in a message group are thus uniquely determined by the message group name, therefore the model user need not type a process name when causing action to be taken to simulate the sending or receiving of message text. (Similarly, the sender and receiver of message text over an operator connection are uniquely determined because the model only allows message text to go from the operator to the associated process.)

Each node will need to maintain some information about the other nodes in the network. (It needs to know about remote processes that have requested access to at least one of its resources, and it needs some information about remote resources that have been requested by at least one of its processes.) The model is designed to create a set of node tables (one table for each node in the network) at each node in the network. Each node will use its set of node tables to maintain the information it needs about all the nodes in the network.

Control messages are used by the model to simulate the transmission of most types of internodal messages. When a message must be sent between nodes, the model will cause text to be printed at the model user's terminal giving the model control message number and stating the destination node and what the message represents. At the time the model user would like the destination node to receive the message, he/she must issue a command to the model to receive the associated control message. OBPL's, message text within message groups, and resource allocation messages are all sent between nodes via control messages. This mechanism was selected so that the effect of internodal messages being delivered with varying delays could be simulated. The only internodal message that the model allows to be processed without user intervention is the one that would be associated with the initiating of a message group. There is no need to model the delay of a message for this because the node in which the accepting process of the message group resides must be aware of the initiation before any checks for deadlock involving that message group will be made.

The types of resource allocation messages that may pass between nodes are 1) requests for access to remote database objects, 2) notification that a process has been granted access to a previously requested database object, and 3) notification that a process has released a database object. If the model user enters a process request for a remote database object, the model will block the process and send a control message (representing a remote resource request) to the node in which the desired database object resides. (Since deadlock detection is being modelled, and resource allocation need not be completely simulated, the model first looks across nodes to verify that the requested database object exists before it sends the control message.) After this control message is received and the desired database object can be allocated to the aforementioned process, a control message stating that the process has been allocated the desired resource is sent to the node in which the process resides. When this new control message is received, the process will be awakened. Although the release of database objects is not necessary to test an algorithm for deadlock detection, a command to allow a process to release a single database object was included in the model for debugging purposes. When a process releases a remote database object, a control message is sent to the node in which the database object resides. The model does not simulate the automatic release of all resources controlled by a process at the time the process reaches a commitment point. This is a feature of process and resource management, and is not relevant to the simulation of a deadlock detection algorithm.

In order to create deadlock situations, processes must be able to gain control of some database objects. The model uses a first-in-first-out allocation scheme for database objects. A process will be blocked if 1) it requests any type of access to a database object that has been exclusively assigned to another process, 2) it requests any type of access to a database object which already has other processes waiting for access to it, or 3) it requests exclusive use of a database object and some process currently has access to the desired database object.

In order to adhere to the belief that the model should be as simple as possible, the model, in expanding an OBPL, does not use the decentralized algorithm exactly as described in the previous chapter. In step 10, the branch to step 3 was removed, thus step 11 is always executed after step 10. When step 11 sends an OBPL to the node in which it is already located, further expansion takes place immediately. Steps 1 and 2 then get executed unnecessarily because RX is properly set in step 10, and the state tables have not been changed during the expansion of the OBPL so the last process to be inserted into the OBPL is still waiting for RX. This implementation was chosen to simplify the coding of the function definition algorithm used to expand OBPL's.

Appendix I contains a Data Structure Diagram for the deadlock detection model, plus a description of the entities and relationships shown in the Diagram. Appendix II contains a brief description of all the user visible functions in the model, followed by the PL/I code of the function definition algorithms which define the model.

VII.4 Test Cases run on the Model

Using the model, several deadlock and near deadlock situations were entered to demonstrate various features of the deadlock detection algorithm. A feature of the ADT system allows a user to save a series of commands in a file, and then type "scenario <file name>" to have the commands executed in order. In each of the cases given, after the system was reinitialized, but before the commands specific to each example were executed, the commands in file "demo0" were executed. The files, along with the output that resulted from the commands in the files, appear in Appendix III. The scenarios are well annotated, and it should be noted that commands to the system appear flush with the margin, whereas output from the Deadlock Detection Model is indented.

The deadlocks created range from one involving two processes and two resources located in a single node, to some involving more than five processes or operators and more than four resources located throughout a three node network. By creating the same deadlock, but altering the order in which processes get blocked and the order in which internodal messages are allowed to arrive, it is shown that the number of times the same deadlock is detected depends on how close (in time) some processes in the deadlock get blocked, and on the locations of the various processes and nodes. (The model works properly regardless of the "simultaneous" processing of commands at various nodes.) Appendix III also includes state diagrams for the test cases which appear in that Appendix. For the cases where a deadlock is created, only the final state is drawn (a key to understanding the diagrams is included), whereas for the cases where there is no deadlock, an important interim state is included in addition to the final state.

The restriction stated in Chapter 4 that a process can not gain access to a database object, release it and request it again before reaching a commitment point, was included to rule out the situation that is shown in "demo_bug". (The scenario was included for demonstration purposes only.)

VIII. Suggestions for Further Research

After a deadlock is detected, at least one involved process must be forced to rescind its request for a resource that is controlled by another process involved in the deadlock. Some of the problems involved in breaking a deadlock (in particular when the deadlock is detected using the decentralized algorithm presented in Chapter VI) are discussed below, as are some issues that may lead to modifications in the schemes presented in Chapters V and VI.

VIII.1 The Rollback/Retry Problem

In order to break a deadlock situation, at least one process involved in the deadlock must be selected and be forced to rollback (backup) to a state prior to the time at which it requested access to the resource for which it was waiting when the deadlock was detected. If the algorithm presented in Chapter VI is being used to detect deadlocks, then (due to the restriction that a process cannot release a database object when it is between commitment points) the process selected for rollback must be returned to its most recent commitment point. In rolling back the process, the external effects created since the last process commitment point must be cancelled.

To accomplish this rollback, it is necessary to undo all database object updates that the process performed within the scope of its current commitment unit (the period since its most recent commitment point), and then release all the database objects that were assigned to the process. In addition, all items of message text that were sent by the process in this commitment unit must be taken back, and all items of message text that were received by the process in this commitment unit must be requeued over the proper connections so that they may again be properly received after the process resumes execution. When taking an item of message text back, if it had already been received by the destination process, this destination process must also be rolled back to its most recent commitment point.

Research needs to be performed to determine an efficient method for rolling back a process. It is possible that some constraints may have to be placed upon communicating processes in order to simplify the rollback problem and lessen the amount of information about a process that must be retained between commitment points. Some papers have been published that deal with the problem of rolling back a database to a previous state. (See [4] for one example.)

Use of the deadlock detection algorithm described in Chapter VI can result in the same deadlock being detected more than once. It therefore may be useful to develop a deterministic algorithm for deciding which process should be rolled back, so that additional processes are not rolled back unnecessarily. Note that if OBPL's are created immediately after a process gets blocked, then every deadlock will be detected with an OBPL that contains only the involved processes. Thus even though a process not involved in a particular deadlock may be waiting for access to a resource which has been assigned to a process in the deadlock, no action need be taken when the deadlock is detected using an OBPL which contains more than the involved processes. One possibility is to impose an arbitrary ordering on the nodes in the network, and always rollback a process in the lowest numbered node that is involved in a given deadlock. This method is unfair in the sense that processes in the higher numbered nodes will rarely be forced to rollback to a previous state. Perhaps a fairer method is to attach a cost factor to each process entry in an OBPL. This cost factor will represent the cost (for the associated process) of computation to date in that process commitment unit. The process with the lowest cost factor will be rolled back with the hope that this minimizes the overall network cost of breaking the deadlock. It is also possible that when the same deadlock is detected more than once, it may be cheaper (from the overall network cost viewpoint) to rollback an extra process occasionally, than to add the extra overhead that is needed for the methods mentioned above. This is a topic which needs to be studied further.

Another related topic which can be investigated involves relaxing some of the restrictions dealing with the release of database objects so that a process can be rolled back to a state somewhere between the previous commitment point and the deadlock state. This may involve slight modifications to the algorithm described in Chapter VI, but may be useful because less code will have to be reexecuted after rollback. (It may be particularly worthwhile when a process is executing a section of code where it is sequentially requesting access to several database objects before reading or updating any of them. Thus a partial, and perhaps sufficient rollback could be accomplished by the release of some of the database objects.)

VIII.2 Optimization and Expansion of the Decentralized Algorithm

If OBPL's are created after a process has been blocked for 'X' units of time (with 'X' greater than 0), then it may be possible to occasionally eliminate the need to create an OBPL after a given process has been blocked for 'X' units of time. When a process is inserted into an OBPL before it has been blocked for 'X' units of time, the need to create an OBPL with this process as the first entry is eliminated. (Additionally, the process may be granted access to the desired resource before 'X' units of time have elapsed, also eliminating the need to create an OBPL.) This type of implementation would affect the scheme used to break deadlocks, as there would no longer be the guarantee that each deadlock would be detected with an OBPL that only contains process entries for the involved processes.

A restriction presented in Chapter IV prevents a process from requesting shared access to a database object and then requesting exclusive use of the same database object. It may be possible to allow this situation will little modification to the decentralized algorithm.

The algorithm presented in Chapter VI requires that all resources be uniquely identifiable. It may be desirable in some applications to allow processes to wait for any one of N identical and interchangeable resources. Inclusion of this property would necessitate a change in the use and expansion of OBPL's. Preliminary study shows that it would be necessary to place control of the expansion of an OBPL with one node (which may be different for each OBPL), since notification would be required after it is ascertained that a loop exists in an OBPL or that an active process has been encountered. This notification is needed because there is a deadlock involving N identical resources only if every process that controls one of these resources is involved in a loop in an OBPL. (This is in contrast to the situation where there are N readers of a given database object and a deadlock exists if any one of these readers is involved in a loop in an OBPL.) Rather than passing an OBPL from node to node, the "controlling" node may request other nodes to expand a section of the OBPL and return it to the "controlling" node. Further study is required to determine exactly how the decentralized algorithm can be modified to include the above mentioned feature.

In addition, it may be worthwhile to study the possibilities of allowing human processes to wait for events external to the computer system (i.e. a phone call or a message from a fellow worker, rather than only wait for a message from a given process) and/or the possibilities of allowing a process to wait for more than one resource at a time.

VIII.3 Types and Probability of Deadlock

In order to get a valid estimation of the cost of using the deadlock detection algorithm presented in Chapter VI, it is necessary to get estimations as to how many processes in how many different nodes are typically involved in a deadlock, and how frequently deadlock can be expected to occur. Some research has been performed dealing with the probability of deadlock in a computer system (see [6]), but to this author's knowledge, no work has been performed dealing with the types (i.e. how many processes in how many different nodes) of deadlock that can be expected in a computer network.

VIII.4 Refinement of the Centralized Algorithm

The scheme presented in Chapter V was not studied extensively. It is possible that it can be refined to a point where little, if any, unnecessary processing takes place in order to determine if a deadlock exists. Due to reliability factors and communications delays, it is not recommended that a centralized scheme be used exclusively in a network. However, a hybrid model of the centralized and decentralized algorithms may prove to be more cost effective than the decentralized algorithm alone. This hybrid model could possibly be constructed by using the centralized scheme for small groups of nodes located within a specified distance of each other, and then using the decentralized scheme between the control nodes for each of the groups using the centralized scheme.

IX. Conclusions

The schemes presented in Chapters II and III were designed to be used to help detect process deadlocks in a computer network where the only allowable wait condition is for the availability of database resources. Many systems only allow this type of process wait, so there is a need for algorithms which solve the problems that the schemes of Chapters II and III attack. However, some alterations must be made to the scheme of Chandra, Howe and Karp and to the decentralized scheme of Mahmoud and Riordon before they can be used to solve the problems they address. It seems that these two schemes, when modified, would result in essentially the same algorithm. This new algorithm would require each node's resource tables to be sent to one node in the network, which will then process all the outstanding requests for access to database objects. (In the case of Mahmoud and Riordon's scheme, perhaps each node would still examine all requests.) The major difference from the original schemes is that no resource allocations would be performed without examining the entire network state. (i.e. requests for access by a process to local resources must still wait for information from other nodes) With or without modifications, the two schemes are inefficient in that they require large tables (when the database is locked at the record level) to be passed between the nodes. Additionally, each node must be capable of processing requests which require the presence of every node's tables in that node. This is an undesirable constraint, because it requires minicomputers which serve as nodes within the network to have the capacity to store (in main memory or secondary storage) the entire network state at one time. Although only minor modifications are required to the schemes so that they will work, they may require some major modifications before they can be used in a general scheme for detecting deadlock in all types (i.e. any size computers and any number of nodes) of computer networks.

The two "centralized" schemes presented in Chapters III and V can both result in message bottlenecks at the control node, and if the control node fails, both result in a significant delay while a new control node is established. Additionally, if the network is geographically spread out, there can be an undesirable delay in some cases when a process requests access to a local database object. It is recommended that neither scheme be used exclusively in a network which covers a large (geographically) area or consists of a large number of nodes.

The decentralized algorithm presented in Chapter VI requires each node to only maintain information relating to its processes and resources. Thus the amount of storage required at each node to support the algorithm is proportional to the total size of the system at that node. Additionally, there is little, if any, delay in granting a process access to an available resource.

The size of messages (OBPL's) passed between the nodes is directly proportional to the number of processes involved in a chain, where each process is waiting for a resource controlled by another process in the chain. It is felt that these chains (and therefore OBPL's) each involve only a few processes, and by delaying the creation of OBPL's until after a process has been blocked for 'X' units of time, the number of OBPL's that must be passed between nodes will be minimal. It should be noted that the decentralized algorithm presented in Chapter VI will work regardless of whether or not processes are allowed to wait for messages which must be sent from other processes within the network.

With the optimization feature discussed earlier, the algorithm presented in Chapter VI is efficient and can be use regardless of the size and composition of a computer network.

References

1. Bachman, Charles W.; Bouvard, Jaques; and Reeves, Raymond J.D. "Architecture Definition Technique: It's Objectives, Theory, Process Facilities and Practice", Internal Memorandum, Honeywell Information Systems, Billerica, Mass., November 26, 1975. (An earlier version appeared in the Proceedings of the 1972 ACM SIGFIDET Workshop, November 1972.)

2. Bachman, Charles W. "Data Structure Diagrams", Data Base, A Quarterly of SIGBDP, Vol. 1, No. 2, Summer 1969, pp. 4-10.

3. Chandra, A.N.: Howe, W.G.: and Karp, D.P. "Communication Protocol for Deadlock Detection in Computer Networks", IBM Technical Disclosure Bulletin, Vol. 16, No. 10, March 1974, pp. 3471-3481.

4. Chandy, K. Many; Browne, James C.; Dissly, Charles W.; and Uhrig, Werner R. "Analytic Models for Rollback and Recovery in Data Base Systems", IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, March 1975, pp. 100-110.

5. Coffman, E.G.: Elphick, M.J.; and Shoshani, A. "System Deadlocks", Computing Surveys, Vol. 3, No. 2, June 1971, pp. 67-78.

6. Ellis, Clarence A. "Probabilistic Models of Computer Deadlock", Report #CU-CS-041-74, University of Colorodo, April 1974.

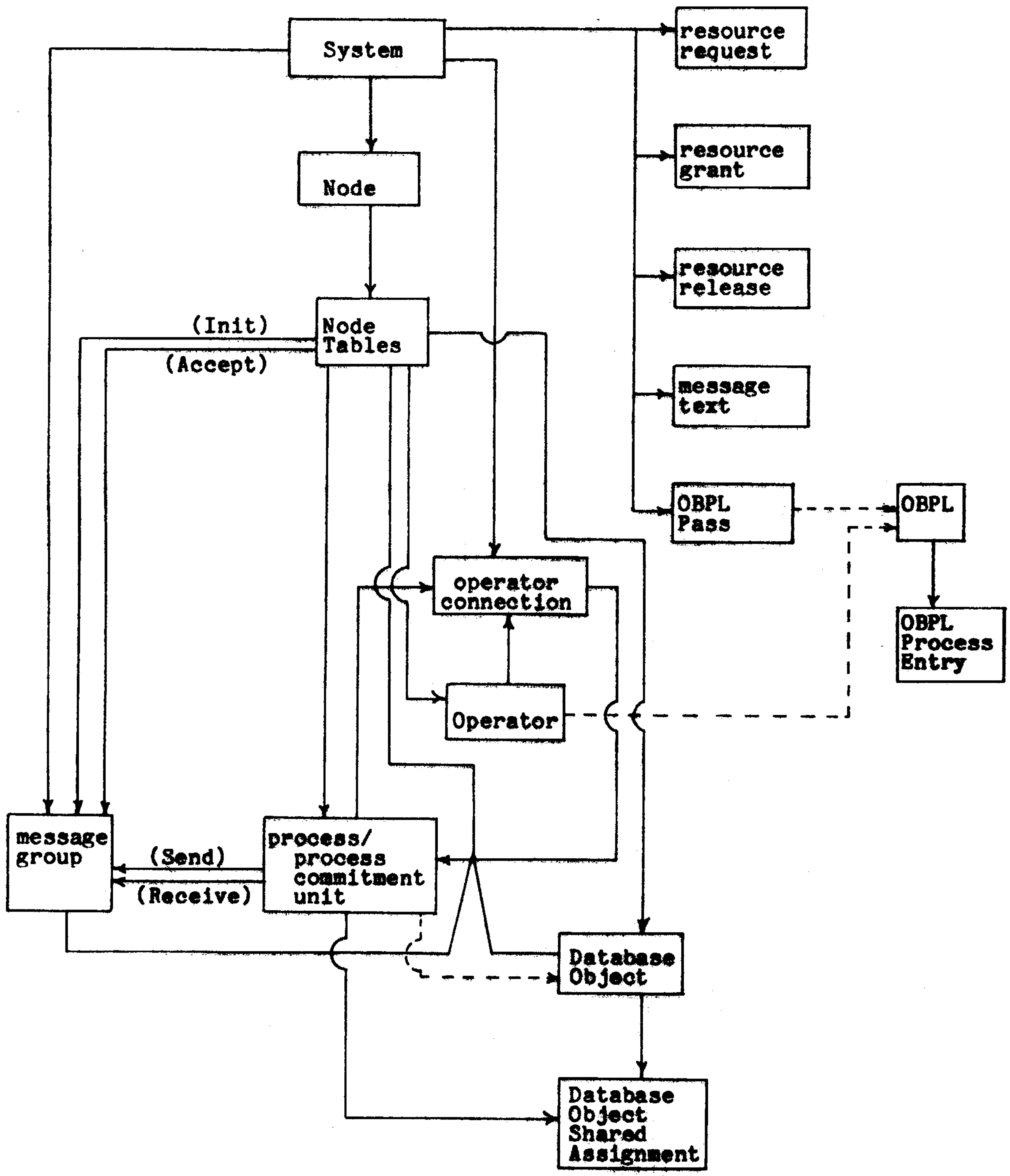
7. Holt, Richard C. "Some Deadlock Properties of Computer Systems", Computing Surveys, Vol. 4, No. 3, September 1972, pp. 179-196.

8. Mahmoud, Samy; and Riordon, J.S. "Protocol Considerations for Software Controlled Access Methods in Distributed Data Bases", Proceedings of the International Symposium on Computer Performance Modeling, Measurement and Evaluation, Harvard University, Cambridge, Mass., March 29-31, 1976, pp. 241-264.

9. Murphy, J.E. "Resource Allocation with System Interlock Detection in a Multitask System", Fall Joint Computer Conference Proceedings, Vol. 33, 1968, pp. 1169-1176.

10. Trinchieri, Mario. "On Managing Interference Caused by Database Sharing", Alta Frequenza, Vol. XLIV, No. 11, 1975, pp. 641-650.

Appendix I



Data Structure Diagram for the ADT Deadlock Detection Model

This section describes the entities which are used in the ADT Deadlock Detection Model. Each entity is described in basically the same manner. The format used is:

```
<ENTITY NAME>
   <text..........
   ...............>

   entity attributes:
      <attribute name>
         <text..........
         ..............>

   entity owner roles:
      <name of set owned by entity>
         <text..........
         ..............>

   entity member roles:
         <name of set where entity is a member>
```

The sets are named in the following way:

owner_name->member_name

Both owner_name and member_name are the names of entities. A qualifier is used to distinguish between two sets which have the same entities as owner and member:

owner_name->member_name(qualifier)

If there are alternate owners or multiple members, the notation used is:

owner_name/owner_name/...->member_name/member_name/... Where attribute names are used, they correspond exactly to the names (which include abbreviations for the entities they represent) that are used in the PL/I code of the Model.

DATABASE_OBJECT

This represents an object within the database which is subject to exclusive (read/write) or shareable (read only) access control. The object may be of various levels of granularity (file, page, record, or item of record). The only requirement is that the entire object is treated uniformly in regard to assignment to a process and subsequent release.

entity attributes:

dbo.name

The unique name for the database object at the node in which it resides.

entity owner roles:

database_object->database_object_shared_assignment

The set of shared_assignment entities for a database object defining the number of processes currently sharing the database object on a read only basis.

database_object->process

The set of processes waiting on the availability of the database object.

(see node_table/dbo/message_group/operator_connection->process)

entity member roles:
node_table->database_object

process->database_object

DATABASE_OBJECT_SHARED_ASSIGNMENT
The mechanism for recording the shared assignment of a database object to a process for read only purposes.

entity_attributes: (none)

entity owner roles: (none)

entity member roles:
database_object->database_object_shared_assignment

process->database_object_shared_assignment

MESSAGE_GROUP
The string of text elements which are sent from one process to another over a specified connection.

entity attributes:
message.name
The network unique name for the message group.

message.number_qd
The number of messages in the message group that have been received by the acceptor of the message group plus the number of messages that are currently queued at the destination end and have not yet been received.

message.number_rcvd
The number of messages in the message group that have been received (read) by the acceptor of the message group.

message.number_sent
The number of messages in the message group that have been sent (regardless of whether or not they have currently reached the destination node) by the initiator of the message group.

entity owner roles:
message_group->process
The set of processes waiting for text in the message group. The nature of exclusive assignment of a message group to a process precludes more than one process to actually be waiting for text.
(see node_table/dbo/message_group/operator_connection->process)

entity member roles:
node_table->message_group(accept)

node_table->message_group(init)

process->message_group(receive)

process->message_group(send)

system->message_group

MESSAGE_TEXT
This represents one message within a message group when the initiator and acceptor are located in different nodes. No actual text need be transmitted, because for the purposes of deadlock detection, the content of the messages is unimportant, and it is only necessary to know how many messages are sent and received.

entity attributes:
msg.mg_name
The message group name to which the "simulated message" belongs.

entity owner roles: (none)

entity member roles:
system->message_text

NODE

A processor in the network which includes a Process Management Module for the purposes of resource allocation and deadlock detection.

entity attributes:
node.name
The network unique name for the node.

entity owner roles:
node->node_table
The set of tables used by a node to maintain all needed information about the nodes in the network.

entity member roles:
system->node

NODE_TABLE

A table used to maintain needed information about operators, processes and resources located at a given node.

entity attributes:
node_table.name
The name of the node about which this table will maintain information.

entity owner roles:
node_table->database_object
The set of database objects located in the node "referenced" by the node table, and for which the node in which the node table resides needs information.

node_table->message_group(accept)
The set of message groups that have been initiated with the accepting process declared to be located in the node which is "referenced" by the node table, and located therein. (If a node table does not "reference" the node in which it is located, then this set is empty for that node table.)

node_table->message_group(init)
The set of message groups that have been initiated by processes located in the node which is "referenced" by the node table, and located therein. (If a node table does not "reference" the node in which it is located, then this set is empty for that node table.)

node_table->operator
The set of operators declared to exist at the node "referenced" by the node table, and for which the node in which the node table resides needs information. (A node only needs to know about the operators at its own node, therefore if a node table noes not "reference" the node in which it is located, this set is empty for that node table.)

node_table->process
The set of processes located in the node "referenced" by the node table, and for which the node in which the node table resides needs information.

node_table/dbo/message_group/operator_connection->process
The set of processes in a particular state. If the owner is a node_table which "references" the node in which it is located, then the process is in the ready or running state. If the owner is a database object, the the process is waiting for access to that database object. If the owner is a message group or operator connection, then the process is waiting for text in that message group or over that operator connection.

entity member roles:
node->node_node_table

OBPL
An ordered blocked process list used to detect deadlock.

entity attributes:
obpl.res_name
The name of the resource for which the most recently inserted process into the OBPL is waiting.

obpl.res_node_name
The name of the node in which the above mentioned resource resides.

obpl.res_type
The type (database object, message in a message group, or message over an operator connection) of the above mentioned resource.

obpl.msg_numb
If the above mentioned resource is a message in a message group, then this attribute contains the number of the message (within the message group) that is being waitied for.

entity owner roles:
OBPL->OBPL_process_entry
The set of processes and operators that have been inserted into the OBPL.

entity member roles:
OBPL_pass->OBPL

operator->OBPL

OBPL_PASS
This is used to pass an OBPL from one node to another, where it can be further expanded.

entity attributes:
obpl_pass.des_node_name
The name of the node to which the OBPL is being sent for further expansion.

entity owner roles:
OBPL_pass->OBPL
This is a one-to-one relationship with the member being the OBPL that is being passed from one node to another.

entity member roles:
system->OBPL_pass

OBPL_PROCESS_ENTRY
This represents a ppocess that has been inserted into an OBPL.

entity attributes:
proc_entry.node_name
The name of the node in which the process that has been entered into the OBPL resides.

proc_entry.process_name
The name of the process that has been entered into the OBPL.

entity owner roles: (none)

entity member roles:
OBPL->OBPL_process_entry

OPERATOR
This entity represents a person that has been declared as an operator at a given node.

entity attributes
operator.name
The unique name for the operator in the node at which he/she is located.

entity owner roles:
operator->OBPL
The set of OBPL's that require state information about the operator before they can be further expanded.

operator->operator_connection
The set of operator connections over which the operator may communicate with processes.

entity member roles:
node_table->operator

OPERATOR_CONNECTION
An entity via which messages are sent from an operator to a process.

entity attributes:
op_con.name
The network unique name for the operator connection

op_con.number_ed
The number of messages that have been sent by the operator but have not yet been received by the process over this operator connection.

entity owner roles:
operator_connection->process
The set of processes waiting for text over the operator connection. The nature of exclusive assignment of an operator connection to a process precludes more than one process to actually be waiting for text.
(see node_table/dbo/message_group/operator_connection->process)

entity member roles:
operator->operator_connection

process->operator_connection

system->operator_connection

PROCESS (PROCESS COMMITMENT UNIT)
This represents a process which is executing within a process commitment unit (the period between process commitment points). Processes are unique, as are process commitment units, therefore the model treats them as one entity.

entity attributes:
process.access_type
If the process is waiting for access to a database object, this attribute denotes the type ("shared" or "exclusive") of access desired.

process.name
The unique name of the process within the node in which it resides.

entity owner roles:
process->dabatase_object
The set of database objects currently exclusively assigned to the process for read/write purposes. If a database object is not inserted in such a set, and its database_object->database_object_shared_assignment set is empty, then it is available for exclusive assignment.

process->database_object_shared_assignment
The set of database_object_shared_assignment entities representing database objects assigned to a process on a shared (read only) basis.

process->message_group(receive)
The set of message groups which have been accepted by the process. (The process can receive messages in these message groups.)

process->message_group(send)
The set of message groups which have been initiated by the process. (The process can send messages in these message groups.)

process->operator_connection
The set of operator connections over which the process can receive messages from operators.

entity member roles:
node_table->process

node_table/dbo/message_group/operator_connection->process

RESOURCE_GRANT
The internodal message granting a process access to a database object located at a different node.

entity attributes:
res_grant.proc_name
The name of the process that is being given access to a database object.

res_grant.proc_nodde_name
The name of the node in which the above mentioned process resides.

res_grant.res_name
The name of the database object which the above mentioned process is gainig access to.

res_grant.res_node_name
The name of the node in which the above mentioned database object resides.

entity owner roles: (none)

entity member roles:
system->resource_grant

RESOURCE_RELEASE
The internodal message stating that a given database object has been released by a specified process.

entity attributes:
res_rel.dest_dbo_name
The name of the database object being released.

res_rel.dest_node_name
The name of the node in which the released database object resides.

res_rel.rel_pnode_name
The name of the node in which the process releasing the database object resides.

res_rel.rel_proc_name
The name of the process releasing the database object.

entity owner roles: (none)

entity member roles:
system->resource_release

RESOURCE_REQUEST
The internodal message in which a process requests access to a database object located at a different node.

entity attributes:
res_req.access_type
The type of access ("shared" or "exclusive") that has been requested.

res_req.dest_dbo_name
The name of the database object to which access has been requested.

res_req.dest_node_name
The name of the node in which the desired database object resides.

res_req.req_node_name
The name of the node in which the requesting process resides.

res_req.req_proc_name
The name of the process requesting access to the above mentioned database object.

entity owner roles: (none)

entity member roles:
system->resource_request

SYSTEM
The computer network.

entity attributes:
system.last_cont_msg
The number of internodal control messages that have been sent in the network.

entity owner roles:
system->message_group
The set of message groups that have been initiated throughout the network.

system->message_text/OBPL/pass/resource_grant/resource_release/resource_request
The set of control messages that have been sent, but have not yet been received by the destination node. The type of control message represented is uniquely determined by the entity type of the member.

system->node
The set of nodes in the network.

system->operator_connection
The set of operator connections that have been declared within the network.

entity member roles: (none)

Appendix II

The ADT Deadlock Detection Model consists of seven PL/I procedures, each of which contains multiple entries. A description of the Deadlock Detection Model user visible functions begins on the next page. Included in the description of a function is the name of the procedure in which that function appears. The seven PL/I procedures follow the function descriptions, and these procedures are followed by the two PL/I include files which are used by the various procedures. File DDM_serv_routines contains declarations of Deadlock Detection Model functions which are called by other functions within the Model, and file ADT_primitives contains declarations of the ADT system functions.

The following is an index to the PL/I procedures and include files.

USER VISIBLE FUNCTIONS
ADT Deadlock Detection Mechanism

acceptmg(p_mg_name, p_accept_node_name, p_accept_proc_name)
Declares process "p_accept_proc_name" located in node "p_accept_node_name" as the only process that can receive messages in the message group specified by "p_mg_name". acceptmg is located within procedure MSG.

cdbo(p_node_name, p_dbo_name)
"Creates" a database object at the node specified by "p_node_name". The database object has a "local" name specified by "p_dbo_name". cdbo is located within procedure DDM.

cnode(p_node_name)
"Creates" a node with the name specified by "p_node_name". cnode is located within procedure DDM.

copcon(p_con_name, p_con_node_name, p_op_name, p_process_name)
"Creates" an operator connection between operator "p_op_name" and process "p_process_name", both located in node "p_con_node_name". The operator connection will have the global name specified by "p_con_name". copcon is located within procedure OP_CON.

cproc(p_node_name, p_process_name)
"Creates" a process with the name specified by "p_process_name" and located in the node specified by "p_node_name". cproc is located within procedure DDM.

dclop(p_op_node_name, p_operator_name)
"Declares" that an operator with name "p_operator_name" exists at the node with name "p_op_node_name". dclop is located within procedure DDM.

initmg(p_mg_name, p_init_node_name, p_init_proc_name, p_accept_node_name)
Declares process "p_init_proc_name" located in node "p_init_node_name" as the only process that can send messages in the message group specified by "p_mg_name". All messages in the message group will be sent to a process in the node specified by "p_accept_node_name". initmg is located within procedure MSG.

opmsg(p_con_name)
"Sends" a message from the operator to the process in operator connection "p_con_name". opmsg is located within procedure OP_CON.

opstat(p_op_node_name, p_op_name, p_state, p_con_name)
States that operator "p_op_name" at node "p_op_node_name" is either "active" or "waiting" (specified by "p_state"). If the operator is waiting, it would like to receive a message from the process in operator connection "p_con_name". opstat is located within procedure OP_CON.

rcvcm(p_cont_msg_numb)
Causes the control message with number specified by "p_cont_msg_numb" to be received by the appropriate node and the required action then takes place. rcvcm is located within procedure RCV_CM.

rcvmsg(p_mg_name)
Causes a message to be "received" in message group "p_mg_name". If no messages are queued, then the receiving process is blocked. rcvmsg is located within procedure MSG.

rcvopmsg(p_con_name)
Causes a message to be "received" by the process in operator connection "p_con_name". If no messages are queued, then the process is blocked and we request the status of the operator involved with this operator connection. rcvopmsg is located within procedure OP_CON.

rldbo(p_proc_node_name, p_process_name, p_dbo_node_name, p_dbo_name)
Causes the database object "p_dbo_name" located in the node specified by "p_dbo_node_name" to be released by process "p_process_name" in node "p_proc_node_name". If additional processes are queued for the database object, they may be removed from the queue in accordance with the rules for resource allocation. rldbo is located within procedure REL.

rqdbo(p_access_type, p_proc_node_name, p_process_name, p_dbo_node_name, p_dbo_name)
Handles a request by process "p_process_name" (located at node "p_proc_node_name") for access ("shared" or "exclusive" as specified by "p_access_type") to the database object specified by "p_dbo_name" and located in the node specified by "p_dbo_node_name". rqdbo is located within procedure REQ.

sendmsg(p_mg_name)
"Sends" a message in the message group specified by "p_mg_name". sendmsg is located within procedure MSG.

sysgen
"Creates" (initializes) the system. sysgen is located within procedure DDM. Internally it also has the name "csys".

```
%;
DDM:  procedure;
        /*  This procedure is a collection of subroutines which either
creates entities needed to model the deadlock detection algorithm proposed
by Barry Goldman or performs services for other routines used in the model.
The following user visible functions are included:
          CREATE DATABASE OBJECT
          CREATE NODE
          CREATE PROCESS
          CREATE SYSTEM
          DECLARE OPERATOR
The following support routines are included:
          DECLARE DATABASE OBJECT
          DECLARE DATABASE OBJECT SHARED ASSIGNMENT
          DECLARE CONTROL MESSAGE
          DECLARE NODE TABLE
          DECLARE OBPL
          DECLARE OBPL CONTROL MESSAGE
          DECLARE PROCESS
          DECLARE PROCESS ENTRY
          DECLARE REMOTE RESOURCE GRANT
          FIND ENTITY LOCATION
          INITIATE OBPL             */
```

```
dcl         cont_msg_numb                   fixed bin;
dcl         dboref                          fixed bin(17);
dcl         eos                             bit(1);
dcl         exp_obpl                        entry(fixed bin(17), char(12));
dcl         message_numb                    fixed bin;
dcl         mgref                           fixed bin(17);
dcl         noderef                         fixed bin(17);
dcl         no_more_nodes                   bit(1);
dcl         obpl_passref                    fixed bin(17);
dcl         obplref                         fixed bin(17);
dcl         opref                           fixed bin(17);
dcl         p_attr_class_name               char(44);
dcl         p_cont_msg_numb                 fixed bin;
dcl         p_dbo_name                      char(*);
dcl         p_dbo_node_name                 char(12);
dcl         p_dcl_cont_msg_numb             fixed bin;
dcl         p_dcl_dbo_name                  char(12);
dcl         p_dcl_entity_class_name         char(20);
dcl         p_dcl_node_tabl_name            char(12);
dcl         p_dcl_proc_name                 char(12);
dcl         p_dcl_ref                       fixed bin (17):
dcl         p_dest_node_name                char(12);
dcl         p_entity_name                   char(12);
dcl         p_entity_ref                    fixed bin(17);
dcl         p_node_name                     char(*);
dcl         p_obplref                       fixed bin(17);
dcl         p_operator_name                 char(*);
dcl         p_op_node_name                  char(*);
dcl         p_ownerref                      fixed bin(17);
dcl         p_process_name                  char(*);
dcl         p_proc_node_name                char(12);
dcl         p_res_name                      char(12);
dcl         p_res_node_name                 char(12);
dcl         p_res_type                      char(7);
dcl         proc_entryref                   fixed bin(17);
dcl         procref                         fixed bin(17);
dcl         proc_termref                    fixed bin(17);
dcl         p_send_node_name                char(12);
dcl         p_set_class_name                char(20);
dcl         res_grant_ref                   fixed bin (17);
dcl         sec_node_name                   char(12);
dcl         sec_noderef                     fixed bin(17):
dcl         tableref                        fixed bin(17);
dcl         temp_name                       char(12);
dcl         temp_ref                        fixed bin(17);
dcl         write_list_                     entry options(variable);
%include ADT_primitives;
```

```
/*        CREATE DATABASE OBJECT                    5/21/76  */
create_database_object:  cdbo:  entry(p_node_name, p_dbo_name);
if find_entity_loc(noderef, "sys->node", SYS_REF, p_node_name, "node.name")
     then do;
          call write_list_("Invalid node name.  ", p_node_name,
                    "does not exist.");
          return;
          end;
eos = find_entity_loc(tableref, "node->node_table", noderef, p_node_name,
          "node_table.name");
if ^ find_entity_loc(dboref, "node->dbo", tableref, p_dbo_name, "dbo.name")
     then do;
          call write_list_("Duplicate database object name");
          return;
          end;
call dcl_dbo(dboref, p_dbo_name);
call insert_(dboref, "node->dbo", "first", tableref);
call write_list_("Database object  ", p_dbo_name, "  created in node  ",
          p_node_name);
return;



/*        CREATE NODE                               5/19/76  */
create_node:  cnode:  entry(p_node_name);
if ^ owner_(SYS_REF, "sys->node")
     then do;
          call write_list_("Illegal request, system has not been created.");
          return;
          end;
call find_first_(noderef, "sys->node", SYS_REF, no_more_nodes);
do while (^ no_more_nodes);
     if extract_(noderef, "node.name") = p_node_name
          then do;
               call write_list_("Duplicate node name");
               return;
               end;
     call find_next_(noderef, "sys->node", no_more_nodes);
     end;
call create_entity_(noderef, "node");
call create_attribute_(noderef, "node.name", "field", 12, p_node_name);
call create_relationship_(noderef, "sys->node", "member");
call insert_(noderef, "sys->node", "first", SYS_REF);
call create_relationship_(noderef, "node->node_table", "owner");
call dcl_node_table(tableref, p_node_name);
call insert_(tableref, "node->node_table", "first", noderef);
          /* We will now  make this new node "aware" of the existence of all
          other nodes, and make all other nodes "aware" of this new node. */
sec_noderef = noderef;
call find_next_(sec_noderef, "sys->node", no_more_nodes);
do while (^ no_more_nodes);
     /*  First create a table entity for the new node to be used by
         another node. */
     call dcl_node_table(tableref, p_node_name);
     call insert_(tableref, "node->node_table", "first", sec_noderef);
     /* Now create a table entity for an existing node to be used by the
        new node. */
     sec_node_name = extract_(sec_noderef, "node.name");
     call dcl_node_table(tableref, sec_node_name);
     call insert_(tableref, "node->node_table", "first", noderef);
     call find_next_(sec_noderef, "sys->node", no_more_nodes);
     end;
call write_list_("Node created:  ", p_node_name);
return;
```

```
/*         CREATE  PROCESS                          5/21/76  */
create_process:  cproc:  entry(p_node_name, p_process_name);
if find_entity_loc(noderef, "sys->node", SYS_REF, p_node_name, "node.name")
     then do;
          call write_list_("Invalid node name. ", p_node_name,
                    "does not exist");
          return;
          end;
eos = find_entity_loc(tableref, "node->node_table", noderef, p_node_name,
          "node_table.name");
if ^ find_entity_loc(procref, "node->process", tableref, p_process_name,
          "process.name")
     then do;
          call write_list_("Duplicate process name");
          return;
          end;
/* If an operator with the same name has been declared at the node,
   print an error message and return */
if ^find_entity_loc(opref, "node->operator", tableref, p_process_name,
          "operator.name")
     then do;
          call write_list_(p_process_name, "has been previously declared",
                    "as an operator at node", p_node_name);
          return;
          end;
call dcl_process(procref, p_process_name);
call insert_(procref, "node->process", "first", tableref);
call insert_(procref, "node/dbo/mg->process", "first", tableref);
call write_list_("Process", p_process_name, "created in node", p_node_name);
return;



/*         CREATE SYSTEM                            5/18/76  */
create_syst:  csys:  sysgen:  entry;
if SYS_REF ^= 0
     then do;
          call write_list_("System already created");
          return;
          end;
call create_entity_(SYS_REF, "system");
call create_attribute_(SYS_REF, "system.last_cont_msg", "field", 10, 0);
call create_relationship_(SYS_REF, "sys->node", "owner");
call create_relationship_(SYS_REF, "sys->msg_grp", "owner");
call create_relationship_(SYS_REF, "sys->control_message", "owner");
call create_relationship_(SYS_REF, "sys->message", "owner");
call create_relationship_(SYS_REF, "sys->op_con", "owner");
call write_list_("System created");
return;



/*         DCL DBO                                  5/27/76  */
dcl_dbo:  entry(p_dcl_ref, p_dcl_dbo_name);
/* This procedure creates an entityy for a database object with name specified
   by "p_dcl_dbo_name" and creates the necessary relationships.  A reference
   to the entity is returned via "p_dcl_ref".  */
call create_entity_(p_dcl_ref, "dbo");
call create_attribute_(p_dcl_ref, "dbo.name", "field", 12, p_dcl_dbo_name);
call create_relationship_(p_dcl_ref, "process->dbo", "member");
call create_relationship_(p_dcl_ref, "node->dbo", "member");
call create_relationship_(p_dcl_ref, "dbo->dbo_sh_asmt", "owner");
call create_relationship_(p_dcl_ref, "node/dbo/mg->process", "owner");
return;
```

```
/*         DCL DBO SH ASMT                            5/27/76  */
dcl_dbo_sh_asmt: entry(p_dcl_ref);
/* This procedure creates an entity for a database object shared assignment
        and returns a pointer to it via "p_dcl_ref"  */
call create_entity_(p_dcl_ref, "dbo_sh_asmt");
call create_relationship_(p_dcl_ref, "process->dbo_sh_asmt", "member");
call create_relationship_(p_dcl_ref, "dbo->dbo_sh_asmt", "member"):
return;



/*         DCL CONTROL MESSAGE                        5/27/76  */
dcl_control_message:  entry(p_dcl_ref, p_dcl_entity_class_name,
          p_dcl_cont_msg_numb);
/* This procedure will establish an OBPL, a remote resource request or a
   remote resource release as a control message.  It will generate a control
   message number (which becomes an attribute of the entity specified by
   "p_dcl_ref") and change the "system entity so that it is aware of the
   new control message number.  This control message number is returned via
   "p_dcl_cont_msg_numb"  */
p_dcl_cont_msg_numb = extract_(SYS_REF, "system.last_cont_msg") + 1;
call alter_(SYS_REF, "system.last_cont_msg", p_dcl_cont_msg_numb);
call create_order_(p_dcl_ref, p_dcl_entity_class_name, "control_message");
call create_relationship_(p_dcl_ref, "sys->control_message", "member");
call create_attribute_(p_dcl_ref, "control_message.number", "field", 10,
          p_dcl_cont_msg_numb);
return;



/*         DCL NODE TABLE                             5/27/76  */
dcl_node_table:  entry(p_dcl_ref, p_dcl_node_tabl_name);
/* This procedure will create an entity for a node table and creates the
   necessary relationships.  The entity is also given the name specified by
   "p_dcl_node_tabl_name".  A pointer to the new entity is returned via
   "p_dcl_ref".  */
call create_entity_(p_dcl_ref, "node_table");
call create_attribute_(p_dcl_ref, "node_table.name", "field", 12,
          p_dcl_node_tabl_name);
call create_relationship_(p_dcl_ref, "node->node_table", "member");
call create_relationship_(p_dcl_ref, "node->operator", "owner");
call create_relationship_(p_dcl_ref, "node->process", "owner");
call create_relationship_(p_dcl_ref, "node->dbo", "owner");
call create_relationship_(p_dcl_ref, "node/dbo/mg->process", "owner");
call create_relationship_(p_dcl_ref, "init_node->message", "owner");
call create_relationship_(p_dcl_ref, "accept_node->message", "owner");
return;
```

```
/*          DECLARE OBPL                                6/24/76  */
dcl_obpl:  entry(p_obplref, p_res_node_name, p_res_name, p_res_type);
/* This procedure will create an entity for an OBPL.  Included in this
   entity will be attribute fields which give the name, type and node name
   for the resource that the most recently inserted process into this OBPL
   is waiting for.  The location of the OBPL entity is returned via the
   parameter "p_obplref"  */
call create_entity_(p_obplref, "obpl");
call create_relationship_(p_obplref, "obpl_pass->obpl", "member");
call create_relationship_(p_obplref, "operator->obpl", "member");
call create_relationship_(p_obplref, "obpl->proc_entry", "owner");
call create_attribute_(p_obplref, "obpl.res_name", "field", 12, p_res_name);
call create_attribute_(p_obplref, "obpl.res_type", "field", 7, p_res_type);
call create_attribute_(p_obplref, "obpl.res_node_name", "field", 12,
          p_res_node_name);
/* Create an attribute (to be altered only when the resource is a message)
   to indicate the message number within a message group that is being
   waited for */
call create_attribute_(p_obplref, "obpl.msg_numb", "field", 4, "0");
return;
```

```
/*          DECLARE OBPL CONTROL MESSAGE                6/25/76  */
dcl_obpl_cont_msg:  entry(p_obplref, p_dest_node_name, p_send_node_name);
/* This procedure creates a control message used to pass the OBPL pointed
   to by "p_obplref" from the node specified by "p_send_node_name" the the
   node specified by "p_dest_node_name"  */
call create_entity_(obpl_passref, "obpl_pass");
call create_attribute_(obpl_passref, "obpl_pass.dest_node_name", "field",
          12, p_dest_node_name);
call create_relationship_(obpl_passref, "obpl_pass->obpl", "owner");
/* Insert the OBPL into this control message */
call insert_(p_obplref, "obpl_pass->obpl", "first", obpl_passref);
/* Declare the "obpl_pass" as a control message */
call dcl_control_message_(obpl_passref, "obpl_pass", cont_msg_numb);
call insert_(obpl_passref, "sys->control_message", "last", SYS_REF);
call write_list_("Control message number", cont_msg_numb, "sent from",
          p_send_node_name, "to", p_dest_node_name);
call write_list_("          representing an OBPL");
return;
```

```
/*         DECLARE OPERATOR                              7/13/76  */
dclop:  entry(p_op_node_name, p_operator_name);
/* This procedure will create an entity for an operator with name specified
   by "p_operator_name" and located at the node specified by
   "p_op_node_name" */
/* If the node specified by "p_op_node_name" does not exist, print an
   error message and return */
if find_entity_loc(noderef, "sys->node", SYS_REF, p_op_node_name,
          "node.name")
     then do;
          call write_list_("Inmalid node name:", p_op_node_name,
                    "does not exist");
          return;
          end;
/* Get the location of the node_table for "p_op_node_name" */
eos = find_entity_loc(tableref, "node->node_table", noderef,
          p_op_node_name, "node_table.name");
/* If "p_operator_name" was previously declared as an operator, print an
   error message and return */
if ^find_entity_loc(opref, "node->operator", tableref, p_operator_name,
          "operator.name")
     then do;
          call write_list_(p_operator_name, "has been previously",
                    "declared as an operator at node", p_op_node_name);
          return;
          end;
/* If "p_operator_name" was previously declared as a process, print an
   error message and return */
if ^find_entity_loc(procref, "node->process", tableref, p_operator_name,
          "process.name")
     then do;
          call write_list_(p_operator_name, "has been previously declared",
                    "as a process at node", p_op_node_name);
          return;
          end;
/* Create an entity for the operator and declare the necessary
   relationships and attributes */
call create_entity_(opref, "operator");
call create_attribute_(opref, "operator.name", "field", 12, p_operator_name);
call create_relationship_(opref, "operator->op_con", "owner");
call create_relationship_(opref, "operator->obpl", "owner");
call create_relationship_(opref, "node->operator", "member");
call insert_(opref, "node->operator", "first", tableref);
call write_list_(p_operator_name, "has been declared as an operator",
          "at node", p_op_node_name);
return;
```

```
/*          DCL PROCESS                                   5/27/76  */
dcl_process:  entry(p_dcl_ref, p_dcl_proc_name);
/* This procedure will create an entity for a process, give it the name
   specified by "p_dcl_proc_name" and create the necessary relationships  */
call create_entity_(p_dcl_ref, "process");
call create_attribute_(p_dcl_ref, "process.name", "field", 12,
          p_dcl_proc_name);
call create_attribute_(p_dcl_ref, "process.access_type", "field", 10, "");
call create_relationship_(p_dcl_ref, "node->process", "member");
call create_relationship_(p_dcl_ref, "node/dbo/mg->process", "member");
call create_relationship_(p_dcl_ref, "process->dbo", "owner");
call create_relationship_(p_dcl_ref, "process->dbo_in_segmt", "owner");
call create_relationship_(p_dcl_ref, "process->lcp_dbo", "owner");
call create_relationship_(p_dcl_ref, "send_proc->message", "owner");
call create_relationship_(p_dcl_ref, "rcv_proc->message", "owner");
return;


/*          DECLARE PROCESS ENTRY                         6/25/76  */
dcl_proc_entry: entry(p_obplref, p_proc_node_name, p_process_name);
/* This procedure will create an entity for a process entry in an OBPL.
   The entity will be inserted into the OBPL pointed to by "p_obplref"
   and its process and location of the process will be specified by
   "p_process_name" and "p_proc_node_name", respectively  */
call create_entity_(proc_entryref, "proc_entry");
call create_relationship_(proc_entryref, "obpl->proc_entry", "member");
call create_attribute_(proc_entryref, "proc_entry.process_name", "field",
          12, p_process_name);
call create_attribute_(proc_entryref, "proc_entry.node_name", "field",
          12, p_proc_node_name);
call insert_(proc_entryref, "obpl->proc_entry", "first", p_obplref);
return;


/*          DECLARE REMOTE RESOURCE GRANT 6/17/76  */
dcl_rem_res_grant:  entry(p_dbo_node_name, p_dbo_name, p_proc_node_name,
          p_process_name, p_cont_msg_numb);
/* This procedure will create an entity for a remote resource allocation and
   then declare it as a control message.  The resource represented by
   "p_dbo_name" at the node represented by "p_dbo_node_name" will be
   allocated to the process represented by "p_process_name" at the node
   represented by "p_proc_node_name".  The control message number will be
   returned via the parameter "p_cont_msg_numb".  */
call create_entity_(res_grant_ref, "res_grant");
call create_attribute_(res_grant_ref, "res_grant.res_node_name",
          "field", 12, p_dbo_node_name);
call create_attribute_(res_grant_ref, "res_grant.res_name",
          "field", 12, p_dbo_name);
call create_attribute_(res_grant_ref, "res_grant.proc_node_name",
          "field", 12, p_proc_node_name);
call create_attribute_(res_grant_ref, "res_grant.proc_name",
          "field", 12, p_process_name);
call dcl_control_message(res_grant_ref, "res_grant",
          p_cont_msg_numb);
call insert_(res_grant_ref, "sys->control_message", "last", SYS_REF);
return;
```

```
/*        FIND ENTITY LOCATION                          5/19/76  */
find_entity_loc:  entry(p_entity_ref, p_set_class_name, p_ownerref,
          p_entity_name, p_attr_class_name) returns(bit(1));
/* This procedure determines the database address of the entity with name
   "p_entity_name" (specified by the attribute "p_attr_class_name") which
   is a member of the set occurrence (designated by the parameter
   "p_set_class_name") owned by the record occurrence designated by
   "p_ownerref".

   If the desired named entity does not exist, a true value ("1"b) is
   returned and "p_entity_ref" is unchanged.  Otherwise a false value ("0"b)
   is returned and "p_entity_ref" is updated with the database address of
   the desired entity.  */
call find_first_(temp_ref, p_set_class_name, p_ownerref, eos);
if ^ eos
    then do;
        temp_name = extract_(temp_ref, p_attr_class_name);
        do while (^ eos & (p_entity_name ^= temp_name));
            call find_next_(temp_ref, p_set_class_name, eos);
            if ^ eos
                then temp_name = extract_(temp_ref, p_attr_class_name);
            end;
        end;
if ^ eos then p_entity_ref = temp_ref;
return (eos);



/*        INITIATE OBPL                                 6/25/76  */
initiate_obpl:  entry(p_proc_node_name, p_process_name, p_res_node_name,
          p_res_name, p_res_type);
/* This procedure will initiate the creation and expansion of an OBPL.  The
   first process to be placed on the list is specified by "p_process_name"
   and is located in the node specified by "p_proc_node_name".  The process
   is waiting for the resource specified by "p_res_name" and located in
   the node specified by "p_res_node_name".  The resource type ("dbo" or
   "message") is specified by "p_res_type".  */
/* Create the OBPL entity and have it initialized with the resource and
   process information given by the parameters.  */
call dcl_obpl(obplref, p_res_node_name, p_res_name, p_res_type);
call dcl_proc_entry(obplref, p_proc_node_name, p_process_name);
/* If the process is waiting for a message, then we must find out the
   message number (within the message group) that is desired and put this
   information into the OBPL.  In addition, if the process and the sender
   of the message are in different nodes, then we must send the OBPL to the
   node which initiated the message group rather than try to expand
   the OBPL right away */
if p_res_type = "message"
    then do;
        /* Get the location of the entity for the message group */
        eos = find_entity_loc(mgref, "sys->message", SYS_REF, p_res_name,
                "message.name");
        /* Get the number of the message desired */
        message_numb = extract_(mgref, "message.number_qd") + 1;
        call alter_(obplref, "obpl.msg_numb", message_numb);
        if p_proc_node_name ^= p_res_node_name
            then do;
                call dcl_obpl_cont_msg(obplref, p_res_node_name,
                        p_proc_node_name);
                return;
                end;
        end;
/* Expand the OBPL as much as possible in this node */
call exp_obpl(obplref, p_res_node_name);
return;
end DDM;
```

```
$:
REQ:   procedure;
        /*  This procedure contains the subroutine which allows processes
to request database objects for shared or exclusive use.  The following
user visible function is included:
        REQUEST DATABASE OBJECT         */


dcl     cont_msg_numb                   fixed bin;
dcl     dbo_noderef                     fixed bin(17);
dcl     dboref                          fixed bin(17);
dcl     dbo_tableref                    fixed bin(17);
dcl     eos                             bit(1);
dcl     exc_ownerref                    fixed bin(17);
dcl     ndm_proc_ownerref               fixed bin(17);
dcl     p_access_type                   char(*);
dcl     p_dbo_name                      char(*);
dcl     p_dbo_node_name                 char(*);
dcl     pnoderef                        fixed bin(17);
dcl     p_process_name                  char(*);
dcl     p_proc_node_name                char(*);
dcl     procref                         fixed bin(17);
dcl     ptableref                       fixed bin(17);
dcl     res_req_ref                     fixed bin(17);
dcl     sh_asmtref                      fixed bin(17);
dcl     write_list_                     entry options(variable);
$include DDM_serv_routines;
$include ADT_primitives;
```

```
/*          REQUEST DATABASE OBJECT                    5/26/76  */
request_dbo:  rqdbo:  entry(p_access_type, p_proc_node_name, p_process_name,
          p_dbo_node_name, p_dbo_name);
/* Verify that the node specified by "p_proc_node_name" exists */
if find_entity_loc(pnoderef, "sys->node", SYS_REF, p_proc_node_name,
          "node.name")
     then do;
          call write_list_("Invalid process node name.  ", p_proc_node_name,
                    "does not exist.");
          return;
          end;
/* Verify that the process specified by "p_process_name" exists at node
     "p_proc_node_name" */
eos = find_entity_loc(ptableref, "node->node_table", pnoderef,
          p_proc_node_name, "node_table.name");
if find_entity_loc(procref, "node->process", ptableref, p_process_name,
          "process.name")
     then do;
          call write_list_("Invalid process name", p_process_name, "at node",
                    p_proc_node_name, "does not exist.");
          return:
          end;
/* Verify that access type is "shared" or "exclusive"  */
if (p_access_type ^= "exclusive") & (p_access_type ^= "shared")
     then do;
          call write_list_("Invalid access type, request not processed");
          return;
          end;
/*  Check if the process is blocked */
call find_owner_(ndm_proc_ownerref, "node/dbo/ms->process", procref);
if entity_class_name_(ndm_proc_ownerref) = "node_table"
     then do;
          call write_list_("Invalid request, process", p_process_name,
                    "at node", p_proc_node_name, "is not active. ");
          return:
          end;
/* Check if the process and resource are at the same node.  */
if p_proc_node_name = p_dbo_node_name
     then do;        /* Process and resource are at the same node */
          /*Verify that the database object specified by "p_dbo_name"
                exists at node "p_dbo_node_name"  */
          if find_entity_loc(dboref, "node->dbo", ptableref, p_dbo_name,
                    "dbo.name")
               then do;
                    call write_list_("Invalid database object name.",
                              p_dbo_name, "at node", p_dbo_node_name,
                              "does not exist."):
                    return;
                    end;
          /*Test to see if the dbo has already been assigned to the process*/
          if inserted_(dboref, "process->dbo")
               then do;        /*Check if the process has exclusive control
                                  of the database object  */
                    call find_owner_(exc_ownerref, "process->dbo", dboref);
                    if procref = exc_ownerref
                         then do;
                              call write_list_("Invalid request.  Process",
                                        p_process_name, "at node",
                                        p_proc_node_name, "already has",
                                        "exclusive control of", p_dbo_name,
                                        "at node", p_dbo_node_name);
                              return;
                              end:
                    end:
               else do:
                    /*Check if the process has shared access to the dbo */
                    if ^ empty_intersection_(procref, "process->dbo_sh_asmt",
```

```
                        dboref, "dbo->dbo_sh_asmt")
              then do;
                   call write_list_("Invalid request.  Process",
                             p_process_name, "at node",
                             p_proc_node_name, "already has",
                             "access to", p_dbo_name, "at node",
                             p_dbo_node_name);
                   return;
                   end;
          end;
/* Check if the database object might be available for assignment */
if inserted_(dboref, "process->dbo") | ^ empty_(dboref,
          "node/dbo/mg->process")
     then do;
          /*Block the process if the database object has been
            assigned to another process for exclusive use or
            if other processes are currently queued for the
            database object  */
          call alter_(procref, "process.access_type",
                    p_access_type);
          call remove_(procref, "node/dbo/mg->process");
          call insert_(procref, "node/dbo/mg->process", "last",
                    dboref);
          call write_list_("Resource not available,",
                    "process blocked.");
          call initiate_obpl(p_proc_node_name, p_process_name,
                    p_dbo_node_name, p_dbo_name, "dbo");
          return;
          end;
/* Check if the request is for shared access */
if p_access_type = "shared"
     then do;       /*Give the process shared access to the desired
                        database object  */
          call del_dbo_sh_asmt(sh_asmtref);
          call insert_(sh_asmtref, "dbo->dbo_sh_asmt", "first",
                    dboref);
          call insert_(sh_asmtref, "process->dbo_sh_asmt", "first",
                    procref);
          call write_list_(p_process_name, "at node",
                    p_proc_node_name, "granted shared access to",
                    p_dbo_name, "at node", p_dbo_node_name);
          return;
          end;
/*The next if statement will be executed if the request is for
       exclusive use of the database object  */
/* Check if any process has shared access to the desired database
   object */
if ^ empty_(dboref, "dbo->dbo_sh_asmt")
     then do;
          /*Queue the process for exclusive use of the database
            object because at least one other process currently
            has shared access to the database object.  */
          call alter_(procref, "process.access_type", "exclusive");
          call remove_(procref, "node/dbo/mg->process");
          call insert_(procref, "node/dbo/mg->process", "last",
                    dboref);
          call write_list_("Resource is not currently available",
                    "for exclusive use, process", p_process_name);
          call write_list_("          at node", p_proc_node_name,
                    "is blocked.");
          call initiate_obpl(p_proc_node_name, p_process_name,
                    p_dbo_node_name, p_dbo_name, "dbo");
          return;
          end;
     else do;       /*Grant the process exclusive use of the desired
                        database object.  */
          call insert_(dboref, "process->dbo", "first", procref);
```

```
                    call write_list_(p_process_name, "at node",
                            p_proc_node_name, "is granted exclusive use",
                            "of", p_dbo_name, "at node", p_dbo_node_name);
                    return;
                    end;
            end;
/* The next section will be executed when a process requests a remote
   resource */
/* Verify that the desired database object exists */
if find_entity_loc(dbo_noderef, "sys->node", SYS_REF, p_dbo_node_name,
        "node.name")
    then do;
        call write_list_("Invalid database object node name. ",
                p_dbo_node_name, "does not exist.");
        return;
        end;
eos = find_entity_loc(dbo_tableref, "node->node_table", dbo_noderef,
        p_dbo_node_name, "node_table.name");
if find_entity_loc(dboref, "node->dbo", dbo_tableref, p_dbo_name, "dbo.name")
    then do;
        call write_list_("Invalid database object name. ", p_dbo_name,
                "at node", p_dbo_node_name, "does not exist.");
        return;
        end;
eos = find_entity_loc(dbo_tableref, "node->node_table", pnoderef,
        p_dbo_node_name, "node_table.name");
/* Check if the node containing the process is aware of the existence of
      the desired database object.  */
if find_entity_loc(dboref, "node->dbo", dbo_tableref, p_dbo_name, "dbo.name")
    then do;        /* Create local information about the remote resource and
                       block the process.  */
        call dcl_dbo(dboref, p_dbo_name);
        call insert_(dboref, "node->dbo", "first", dbo_tableref);
        call alter_(procref, "process.access_type", p_access_type);
        call remove_(procref, "node/dbo/mg->process");
        call insert_(procref, "node/dbo/mg->process", "last", dboref);
        end;
    else do;        /* Check if the database object has already been assigned
                       to the process.  If it has, print an error message,
                       otherwise block the process.  */
        if inserted_(dboref, "process->dbo")
            then do;
                call find_owner_(exc_ownerref, "process->dbo", dboref);
                if procref = exc_ownerref
                    then do;
                        call write_list_("Invalid request.  Process",
                                p_process_name, "at node",
                                p_proc_node_name, "already has",
                                "exclusive control of", p_dbo_name,
                                "at node", p_dbo_node_name);
                        return;
                        end;
                end;
            else do;
                if ^ empty_intersection_(procref, "process->dbo_sh_asmt",
                        dboref, "dbo->dbo_sh_asmt")
                    then do;
                        call write_list_("Invalid request.  Process",
                                p_process_name, "at node",
                                p_proc_node_name, "already has",
                                "shared access to", p_dbo_name,
                                "at node", p_dbo_node_name);
                        return;
                        end;
                end;
        /* Legal request, "block" the process.  */
        call alter_(procref, "process.access_type", p_access_type);
```

```
          call remove_(procref, "node/dbo/mg->process"):
          call insert_(procref, "node/dbo/mg->process", "last", dboref);
          end;
call write_list_("Process", p_process_name, "at node", p_proc_node_name,
          "is blocked while a request is sent to");
call write_list_("          the node containing the desired resource");
/* Create an entity for a remote resource request and then declare it
        as a control message  */
call create_entity_(res_req_ref, "res_req");
call create_attribute_(res_req_ref, "res_req.access_type", "field", 9,
          p_access_type);
call create_attribute_(res_req_ref, "res_req.req_node_name", "field", 12,
          p_proc_node_name);
call create_attribute_(res_req_ref, "res_req.req_proc_name", "field", 12,
          p_process_name);
call create_attribute_(res_req_ref, "res_req.dest_node_name", "field", 12,
          p_dbo_node_name);
call create_attribute_(res_req_ref, "res_req.dest_dbo_name", "field", 12,
          p_dbo_name);
call dcl_control_message(res_req_ref, "res_req", cont_msg_numb);
call insert_(res_req_ref, "sys->control_messages", "last", SYS_REF);
call write_list_("Control message number", cont_msg_numb, "sent from",
          p_proc_node_name, "to", p_dbo_node_name);
call write_list_("          representing a remote resource request");
return;
end REQ:
```

```
%.
MSG: procedure;
          /*  This procedure contains the subroutines which perform the
message management functions for process to process communication within
a network.  The following user visible functions are included:
          ACCEPT MESSAGE GROUP
          INITIATE MESSAGE GROUP
          RECEIVE MESSAGE
          SEND MESSAGE                        */


dcl       accept_node_name                    char(12);
dcl       accept_node_tableref                fixed bin(17);
dcl       accept_proc_name                    char(12);
dcl       accept_procref                      fixed bin(17);
dcl       cont_msg_numb                       fixed bin;
dcl       eos                                 bit(1);
dcl       init_node_name                      char(12);
dcl       init_node_tableref                  fixed bin(17);
dcl       init_proc_name                      char(12);
dcl       init_procref                        fixed bin(17);
dcl       messageref                          fixed bin(17);
dcl       mgref                               fixed bin(17);
dcl       ndm_proc_ownerref                   fixed bin(17);
dcl       noderef                             fixed bin(17);
dcl       p_accept_node_name                  char(*);
dcl       p_accept_proc_name                  char(*);
dcl       p_init_node_name                    char(*);
dcl       p_init_proc_name                    char(*);
dcl       p_mg_name                           char(*);
dcl       procref                             fixed bin(17);
dcl       rcv_msg_numb                        fixed bin;
dcl       send_msg_numb                       fixed bin;
dcl       write_list_                         entry options(variable);
%include DDM_serv_routines;
%include ADT_primitives;
```

```
/*         ACCEPT MESsAGE GROUP                          7/1/76  */
acceptmg:  entry(p_mg_name, p_accept_node_name, p_accept_proc_name);
/* After this procedure is executed, the process specified by
   "p_accept_proc_name" (and located at the node specified by
   "p_accept_node_name") will be able to accept messages in the message
   group specified by "p_mg_name" */
/* If the message group specified by "p_mg_name" does not exist, print
   an error message and return */
if find_entity_loc(mgref, "sys->message", SYS_REP, p_mg_name,
         "message.name")
    then do;
         call write_list_("Invalid message group name:  ", p_mg_name,
                  " does not exist");
         return;
         end;
/* If the message group has already been accepted by a process, print an
   error message and return */
if inserted_(mgref, "rcv_proc->message")
    then do;
         call write_list_("Invalid accept message group.  ", p_mg_name,
                  " has already been accepted by a process");
         return;
         end;
/* If the node specified by "p_accept_node_name" is not the accepting
   node that was specified when the message group was initialized,
   print an error message and return */
call find_owner_(accept_node_tableref, "accept_node->message", mgref);
if p_accept_node_name ^= extract_(accept_node_tableref, "node_table.name")
    then do;
         call write_list_(p_accept_node_name, " is not the node that was ",
                  "specified to accept ", p_mg_name, " when the message");
         call write_list_("         group was initialized.  The acceptmg",
                  " request is rejected");
         return;
         end;
/* If the process specified by "p_accept_proc_name" does not exist
   at  the node specified by "p_accept_node_name", print an error
   message  and return */
if find_entity_loc(accept_procref, "node->process", accept_node_tableref,
         p_accept_proc_name, "process.name")
    then do;
         call write_list_("Invalid process name:   ", p_accept_proc_name,
                  "does not exist at node ", p_accept_node_name);
         return;
         end;
/* If the process accepting the message group is not active, print an error
   message and return */
call find_owner_(ndm_proc_ownerref, "node/dbo/mg->process", accept_procref);
if entity_class_name_(ndm_proc_ownerref) ^= "node_table"
    then do;
         call write_list_("Invalid acceptmg command.  Process",
                  p_accept_proc_name, "is not active");
         return;
         end;
/* If the process accepting the message group is the same one that
   initiated it, print an error message and return */
call find_owner_(init_node_tableref, "init_node->message", mgref);
if init_node_tableref = accept_node_tableref
    then do;
         /* The initiating and accepting nodes are the same.  See if
            the initiating and accepting processes are the same */
         call find_owner_(init_procref, "send_proc->message", mgref);
         if init_procref = accept_procref
              then do;
                   call write_list_("Initiating and accepting processes",
                            "are the same for message group ", p_mg_name,
                            "acceptmg command rejected");
```

```
                    return;
                    end;
          end;
/* Insert the message group entity into the accept set for the process
   specified by "p_accept_proc_name" */
call insert_(mgref, "rcv_proc->message", "first", accept_procref);
call write_list_(p_mg_name, " has been accepted by ", p_accept_proc_name,
          " at node ", p_accept_node_name);
return;


/*        INITIATE MESSAGE GROUP                    7/1/76  */
initmg:   entry(p_mg_name, p_init_node_name, p_init_proc_name,
          p_accept_node_name);
/* This procedure will create a message group with the global name
   specified by "p_mg_name".  The only  process that can send messages
   in this message group is specified by "p_init_proc_name" and is located
   at the node specified by "p_init_node_name".  The process that will
   receive messages in the given message group is located in the
   node specified by "p_accept_node_name".  The specific process that will
   accept the messages will be given in a subsequent call to "acceptmg"
   by the user.  */
/* If we have a duplicate message group name, we must  print an error
   message and return */
if ^find_entity_loc(mgref, "sys->message", SYS_REF, p_mg_name, "message.name")
     then do;
          call write_list_("Duplicate message group name.  initmg",
                    "command rejected");
          return;
          end;
/* If the node specified by "p_init_node_name" does not exist, print
   and error message and return */
if find_entity_loc(noderef, "sys->node", SYS_REF, p_init_node_name,
          "node.name")
     then do;
          call write_list_("Invalid node name:  ", p_init_node_name,
                    "  does not exist");
          return;
          end;
/* Get the location of the node_table for "p_init_node_name" */
eos = find_entity_loc(init_node_tableref, "node->node_table", noderef,
          p_init_node_name, "node_table.name");
/* If the  process specified by "p_init_proc_name" does not exist at the
   node specified by "p_init_node_name", then print an error message
   and return */
if find_entity_loc(procref, "node->process", init_node_tableref,
          p_init_proc_name, "process.name")
     then do;
          call write_list_("Invalid process name:  ", p_init_proc_name,
                    " does not exist at ", p_init_node_name);
          return;
          end;
/* If the process specified by "p_init_proc_name" is not active, print
   an error message and return */
call find_owner_(ndm_proc_ownerref, "node/dbo/mg->process", procref);
if entity_class_name_(ndm_proc_ownerref) ^= "node_table"
     then do;
          call write_list_("Invalid initmg command.  Process ",
                    p_init_proc_name, " is not active");
          return;
          end;
/* If the node specified by "p_accept_node_name" does not exist, print
   an error message and return */
if find_entity_loc(noderef, "sys->node", SYS_REF, p_accept_node_name,
          "node.name")
```

```
     then do;
          call write_list_("Invalid node name:  ", p_accept_node_name,
                    " does not exist"):
          return;
          end;
/* Get the location of the node_table for "p_accept_node_name" */
eos = find_entity_loc(accept_node_tableref, "node->node_table", noderef,
          p_accept_node_name, "node_table.name");
/* Create  an entity for a message group, create the necessary relationships
   and attributes, and insert the entity into the appropriate sets */
call create_entity_(mgref, "message");
call create_relationship_(mgref, "sys->message", "member");
call create_relationship_(mgref, "init_node->message", "member");
call create_relationship_(mgref, "accept_node->message", "member");
call create_relationship_(mgref, "send_proc->message", "member");
call create_relationship_(mgref, "rcv_proc->message", "member");
call create_relationship_(mgref, "node/dbo/mg->process", "owner");
call create_attribute_(mgref, "message.number_sent", "field", 4, "0");
call create_attribute_(mgref, "message.number_qd", "field", 4, "0");
call create_attribute_(mgref, "message.number_rcvd", "field", 4, "0");
call create_attribute_(mgref, "message.name", "field", 12,  p_mg_name);
call insert_(mgref, "sys->message", "first", SYS_REF);
call insert_(mgref, "init_node->message", "first", init_node_tableref);
call insert_(mgref, "accept_node->message", "first", accept_node_tableref):
call insert_(mgref, "send_proc->message", "first", procref);
call write_list_("Message group ", p_mg_name, "has been initiated");
return:
```

```
/*        RECEIVE MESSAGE                          7/1/76  */
rcvmsg:  entry(p_mg_name);
/* this procedure will simulate the receiveing of a message in the message
   group specified by "p_mg_name" */
/* If the message group specified by "p_mg_name" does not  exist, print
   an error message and return */
if find_entity_loc(mgref, "sys->message", SYS_REF, p_mg_name, "message.name")
    then do;
         call write_list_("invalid message group name:  ", p_mg_name,
                    " does not exist");
         return;
         end;
/* If no process has accepted the message group, print an error message
   and return */
if ^inserted_(mgref, "rcv_proc->message")
    then do;
         call write_list_("Invalid rcvmsg command.  No process has",
                    "accepted message group ", p_mg_name);
         return;
         end;
/* Get the name and node of the process that should receive the message */
call find_owner_(accept_procref, "rcv_proc->message", mgref);
accept_proc_name = extract_(accept_procref, "process.name");
call find_owner_(accept_node_tableref, "accept_node->message", mgref);
accept_node_name = extract_(accept_node_tableref, "node_table.name");
/* If the process specified by "accept_proc_name" is not active, print
   an error message and return */
call find_owner_(ndm_proc_ownerref, "node/dbo/mg->process", accept_procref);
if entity_class_name_(ndm_proc_ownerref) = "node_table"
    then do;
         call write_list_("Process", accept_proc_name, "at node",
                    accept_node_name, "is not active.  No message can be");
         call write_list_("          received in message group", p_mg_name);
         return;
         end;
/* Find out if the message can be received, or if the process must
   be blocked */
rcv_msg_numb = extract_(mgref, "message.number_rcvd");
if rcv_msg_numb < extract_(mgref, "message.number_qd")
    then do;
         /* Allow the process to receive the message */
         rcv_msg_numb = rcv_msg_numb + 1;
         call alter_(mgref, "message.number_rcvd", rcv_msg_numb);
         call write_list_("Process", accept_proc_name, "at node",
                    accept_node_name, "has received");
         call write_list_("          a message in message group", p_mg_name);
         return;
         end;
    else do;
         /* Block the process */
         call remove_(accept_procref, "node/dbo/mg->process");
         call insert_(accept_procref, "node/dbo/mg->process", "first",
                    mgref);
         call write_list_("Process ", accept_proc_name, "at node",
                    accept_node_name, "is blocked waiting for a");
         call write_list_("          message in message group", p_mg_name);
         /* Get the name of the node that initiated (or "owns") the message
            group */
         call find_owner_(init_node_tableref, "init_node->message", mgref);
         init_node_name = extract_(init_node_tableref, "node_table.name");
         /* Check for deadlock */
         call initiate_obpl(accept_node_name, accept_proc_name,
                    init_node_name, p_mg_name, "message");
         return;
         end;
```

```
/*        SEND MESSAGE                                  7/1/76  */
sendmsg:  entry(p_mg_name);
/* This procedure will simulate the sending of a message in the message
   group specified by "p_mg_name" */
/* If the  message group specified by "p_mg_name" does not exist, print
   an error message and return */
if find_entity_loc(mgref, "sys->message", SYS_REF,  p_mg_name, "message.name")
     then do;
          call write_list_("Invalid message group name:  ",  p_mg_name,
                    "does not exist");
          return;
          end;
/* Verify that the process that should send the message is active */
call find_owner_(init_procref, "send_proc->message", mgref);
call find_owner_(ndm_proc_ownerref,  "node/dbo/mg->process", init_procref);
if entity_class_name_(ndm_proc_ownerref) ^= "node_table"
     then do;
          /* The process that should send the message is not active.  Get
             its name and node and print an error message. */
          call find_owner_(init_node_tableref, "init_node->message", mgref);
          init_node_name = extract_(init_node_tableref, "node_table.name");
          init_proc_name = extract_(init_procref, "process.name");
          call write_list_("Process ", init_proc_name, " at node ",
                    init_node_name, " is not active.  No message can be ");
          call write_list_("          sent in message group ", p_mg_name);
          return;
          end;
/* Add 1 to the number of messages sent in this message group */
send_msg_numb = extract_(mgref, "message.number_sent") + 1;
call alter_(mgref, "message.number_sent", send_msg_numb);
/* Find out if the message must be sent between nodes */
call find_owner_(init_node_tableref, "init_node->message", mgref);
call find_owner_(accept_node_tableref, "accept_node->message", mgref);
if init_node_tableref ^= accept_node_tableref
     then do;
          /* Send a control message stating that a message has been sent in
             the message group specified by "p_mg_name" */
          call create_entity_(messageref, "msg");
          call create_attribute_(messageref, "msg.mg_name", "field", 12,
                    p_mg_name);
          call dcl_control_message(messageref, "msg", cont_msg_numb);
          call insert_(messageref, "sys->control_message", "last", SYS_REF);
          /* Get the names of the nodes involved */
          init_node_name = extract_(init_node_tableref, "node_table.name");
          accept_node_name = extract_(accept_node_tableref,
                    "node_table.name");
          call write_list_("Control message number ", cont_msg_numb,
                    " sent from ", init_node_name, " to ", accept_node_name);
          call write_list_("          representing a message in a message",
                    "group");
          return;
          end;
/* If the next section of code is executed, then the message  should be sent
   between processes at the same node */
/* The number of messages queued equals the number of messages sent because
   there is no delay across any node */
call alter_(mgref, "message.number_qd", send_msg_numb);
call write_list_("A message has been sent in message group ", p_mg_name);
/* If no process has accepted the message group, return rather than see if
   a process should be woken up */
if ^inserted_(mgref, "rcv_proc->message")
     then return;
/* If a process is waiting for this message, wake it up and let it "receive"
   the message */
call find_owner_(accept_procref, "rcv_proc->message",  mgref);
call find_owner_(ndm_proc_ownerref, "node/dbo/mg->process", accept_procref);
if ndm_proc_ownerref = mgref
```

```
     then do;
          /* Wake up the process pointed to by "accept_procref" */
          call remove_(accept_procref, "node/dbo/mg->process");
          call insert_(accept_procref, "node/dbo/mg->process", "first",
                    accept_node_tableref);
          /* "Receive" the message */
          rcv_msg_numb = extract_(mgref, "message.number_rcvd") + 1;
          call alter_(mgref, "message.number_rcvd", rcv_msg_numb);
          /* Get the name of the process that was awakened */
          accept_node_name = extract_(accept_node_tableref,
                    "node_table.name");
          accept_proc_name = extract_(accept_procref, "process.name");
          call write_list_("Process", accept_proc_name, "at node",
                    accept_node_name, "has been awakened upon");
          call write_list_("        receipt of a message in message group",
                    p_mg_name);
          end;
return;
end MSG;
```

```
%:
OP_CON:  procedure;
/* This procedure contains subroutines which create an operator connection,
   allow the operator to send messages over the connection, allow the
   operator to receive messages over the connection, and allow the
   operator to report its status (active or blocked) with respect to the
   operator connection.  The following user visible functions are included:
          CREATE OPERATOR CONNECTION
          OPERATOR MESSAGE
          OPERATOR STATUS
          RECEIVE OPERATOR MESSAGE      */



dcl       con_opref                      fixed bin(17);
dcl       eos                            bit(1);
dcl       ndm_proc_ownerref              fixed bin(17);
dcl       noderef                        fixed bin(17);
dcl       node_tableref                  fixed bin(17);
dcl       number_qd                      fixed bin;
dcl       obplref                        fixed bin(17);
dcl       op_conref                      fixed bin(17);
dcl       opref                          fixed bin(17);
dcl       p_con_name                     char(*);
dcl       p_con_node_name                char(*);
dcl       p_op_name                      char(*);
dcl       p_op_node_name                 char(*);
dcl       p_process_name                 char(*);
dcl       process_name                   char(12);
dcl       proc_node_name                 char(12);
dcl       procref                        fixed bin(17);
dcl       p_state                        char(*);
dcl       tableref                       fixed bin(17);
dcl       write_list_                    entry options(variable);
%include DDM_serv_routines;
%include ADT_primitives;



/*        CREATE OPERATOR CONNECTION               7/9/76  */
copcon:  entry(p_con_name, p_con_node_name, p_op_name, p_process_name);
/* This procedure will create a connection between the operator specified
   by "p_op_name" and the process specified by "p_process_name", both
   located at the node specified by "p_op_node_name".  The connection will
   be given the name specified by "p_con_name"  */
/* If we have a duplicate operator connection name, print an error message
   and return */
if ^find_entity_loc(op_conref, "sys->op_con", SYS_REF, p_con_name,
          "op_con.name")
     then do;
          call write_list_("Duplicate operator connection name.",
                    "Command rejected");
          return;
          end;
/* If the node specified by "p_con_node_name" does not exist, print an
   error message and return */
if find_entity_loc(noderef, "sys->node", SYS_REF, p_con_node_name,
          "node.name")
     then do;
          call write_list_("Invalid node name:", p_con_node_name,
                    "does not exist");
          return;
          end;
/* Get the location of the node_table for "p_con_node_name" */
eos = find_entity_loc(node_tableref, "node->node_table", noderef,
```

```
          p_con_node_name, "node_table.name"):
/* If the node is unaware of the existence of the operator, print an
   error message and return */
if find_entity_loc(opref, "node->operator", node_tableref, p_op_name,
          "operator.name")
     then do;
          call write_list_("Invalid operator name:", p_op_name,
                    "does not exist at node", p_con_node_name);
          return;
          end;
/* If the process specified by "p_process_name" does not exist at the
   node specified by "p_con_node_name", print an error message and return */
if find_entity_loc(procref, "node->process", node_tableref,
          p_process_name, "process.name")
     then do;
          call write_list_("Invalid process name:", p_process_name,
                    "does not exist at node", p_con_node_name);
          return;
          end;
/* If the process specified by "p_process_name" is not active, print an
   error message and return */
call find_owner_(ndm_proc_ownerref, "node/dbo/mg->process", procref);
if entity_class_name_(ndm_proc_ownerref) ^= "node_table"
     then do;
          call write_list_("Invalid copcon command.  Process", p_process_name,
                    "is not active");
          return;
          end;
/* Create an entity for an operator connection and insert it into the
   proper sets */
call create_entity_(op_conref, "op_con");
call create_attribute_(op_conref, "op_con.name", "field", 12, p_con_name);
call create_attribute_(op_conref, "op_con.number_qd", "field", 4, "0");
call create_relationship_(op_conref, "process->op_con", "member");
call create_relationship_(op_conref, "operator->op_con", "member");
call create_relationship_(op_conref, "sys->op_con", "member");
call create_relationship_(op_conref, "node/dbo/mg->process", "owner");
call insert_(op_conref, "process->op_con", "first", procref);
call insert_(op_conref, "operator->op_con", "first", opref);
call insert_(op_conref, "sys->op_con", "first", SYS_REF);
call write_list_("Operator connection", p_con_name, "has been established");
return;
```

```
/*        OPERATOR MESSAGE                          7/13/76  */
opmsg:  entry(p_con_name);
/* This procedure will cause a message to be sent from an operator to a
   process over the operator connection specified by "p_con_name".  If a
   process is waiting for this message, it will be awakened and given
   the message, otherwise the message will be queued.  Any OBPL's that
   were waiting for state information about the operator with respect to
   this operator connection will be discarded since the operator is active */
/* If the operator connection specified by "p_con_name" does not exist,
   print an error message and return */
if find_entity_loc(op_conref, "sys->op_con", SYS_REF, p_con_name,
          "op_con.name")
     then do;
          call write_list_("Invalid operator connection name:",
                    p_con_name, "does not exist");
          return;
          end;
/* Discard any OBPL's that were waiting for state information from the
   operator that sent the message */
call find_owner_(opref, "operator->op_con", op_conref);
call find_first_(obplref, "operator->obpl", opref, eos);
do while (¬eos);
     call remove_(obplref, "operator->obpl");
     call find_first_(obplref, "operator->obpl", opref, eos);
     end;
/* If no process is waiting for the message, queue it an return */
if empty_(op_conref, "node/dbo/mg->process")
     then do;
          number_qd = extract_(op_conref, "op_con.number_qd") + 1;
          call alter_(op_conref, "op_con.number_qd", number_qd);
          call write_list_("No process is waiting for the message,",
                    "so it is queued");
          return;
          end;
/* A process is waiting for the message, so we must wake it up */
call find_first_(procref, "node/dbo/mg->process", op_conref, eos);
call remove_(procref, "node/dbo/mg->process");
call find_owner_(tableref, "node->process", procref);
call insert_(procref, "node/dbo/mg->process", "first", tableref);
/* Get the name of the process that was awakened */
process_name = extract_(procref, "process.name");
proc_node_name = extract_(tableref, "node_table.name");
call write_list_(process_name, "at node", proc_node_name, "has been",
          "awakened upon");
call write_list_("         receipt of a message over operator connection",
          p_con_name);
return;
```

```
/*        OPERATOR STATUS                              7/14/76  */
opstat:  entry(p_op_node_name, p_op_name, p_state, p_con_name);
/* This procedure will take the appropriate action when an operator
   reports that it is "active" or "waiting" */
/* If the node specified by "p_op_node_name" does not exist, print an
   error message and return */
if find_entity_loc(noderef, "sys->node", SYS_REF, p_op_node_name,
          "node.name")
     then do;
          call write_list_("Invalid node name:", p_op_node_name,
                    "does not exist");
          return;
          end;
/* Get the location of the node table for the node specified by
   "p_op_node_name" */
eos = find_entity_loc(tableref, "node->node_table", noderef,
          p_op_node_name, "node_table.name");
/* if the operator specified by "p_op_name" does not exist at the given
   node, print an error message and return */
if find_entity_loc(opref, "node->operator", tableref, p_op_name,
          "operator.name")
     then do;
          call write_list_("Invalid operator name:", p_op_name,
                    "does not exist");
          return;
          end;
/* If the operator is active, we can discard all OBPL's that desired this
   state information, and then return */
if p_state = "active"
     then do;
          call find_first_(obplref, "operator->obpl", opref, eos);
          do while (^eos);
               call remove_(obplref, "operator->obpl");
               call find_first_(obplref, "operator->obpl", opref, eos);
               end;
          call write_list_("All OBPL's waiting for the given state",
                    "information have been discarded");
          return;
          end;
if p_state ^= "waiting"
     then do;
          call write_list_("Invalid state.  An operator can only be",
                    "active or waiting");
          return;
          end;
/* If the operator connection specified by "p_con_name" does not exist,
   print an error message and return because one can not wait for a
   message over a non_existent operator connection */
if find_entity_loc(op_conref, "sys->op_con", SYS_REF, p_con_name,
          "op_con.name")
     then do;
          call write_list_("Invalid operator connection name:",
                    p_con_name, "does not exist");
          return;
          end;
/* If the operator specified  by "p_op_name" is not involved with the
   operator connection specified by "p_con_name", print an error message
   and return */
call find_owner_(con_opref, "operator->op_con", op_conref);
if opref ^= con_opref
     then do;
          call write_list_(p_op_name, "at node", p_op_node_name,
                    "is not associated with operator connection",
                    p_con_name);
          return;
          end;
call write_list_("We will now check for deadlock involving the given",
```

```
          "operator");
call write_list_("          and operator connection");
/* If the process that can send messages over the operator connection
   specified by "p_con_name" is active, there is no deadlock, so
   discard all OBPL's that requested the given state information */
call find_owner_(procref, "process->op_con", op_conref);
call find_owner_(ndm_proc_ownerref, "node/dbo/mg->process", procref);
if entity_class_name_(ndm_proc_ownerref) = "node_table"
     then do:
          call find_first_(obplref, "operator->obpl", opref, eos);
          do  while (^eos):
               call remove_(obplref, "operator->obpl");
               call find_first_(obplref, "operator->obpl", opref, eos);
               end;
          return;
          end;
/* If there are no OBPL's waiting for state information about this
   operator, create an OBPL with the operator as the only process entry */
if empty_(opref, "operator->obpl")
     then do;
          call dcl_obpl(obplref, p_op_node_name, "", "op_msg");
          call dcl_proc_entry(obplref, p_op_node_name, p_op_name);
          call insert_(obplref, "operator->obpl", "first", opref);
          end:
/* Find out the name of the process that can send the message the
   operator  desires */
process_name = extract_(procref, "process.name");
/* Expand each OBPL that required state information about the given
   operator */
call find_first_(obplref, "operator->obpl", opref, eos);
do while (^eos);
     /* Remove the OBPL from the set belonging to the given operator */
     call remove_(obplref, "operator->obpl");
     /* Check if we have a deadlock */
     call check_for_deadlock(obplref, p_op_node_name, process_name, eos);
     /* If eos = 1, then a deadlock was not detected, so we should add a
        resource to the OBPL and then expand it */
     if eos
          then do;
               call obpl_add_resource(obplref, ndm_proc_ownerref,
                         p_op_node_name, eos);
               /* If eos = 1 then the resource the process is waiting for
                  is in the same node as the process, so we can continue
                  to expand the OBPL */
               if eos
                    then call exp_obpl(obplref, p_op_node_name);
               end;
     /* See if there are any more OBPL's to be examined */
     call find_first_(obplref, "operator->obpl", opref, eos);
     end;
return;
```

```
/*        RECEIVE OPERATOR MESSAGE                    7/13/76  */
rcvopmsg:  entry(p_con_name);
/* This procedure will simulate the receiving of a message by a process
   over the operator connection specified by "p_con_name" */
/* If the operator connection specified by "p_con_name" does not exist,
   print an error message and return */
if find_entity_loc(op_conref, "sys->op_con", SYS_REF, p_con_name,
      "op_con.name")
     then do;
          call write_list_("Invalid operator connection name:", p_con_name,
                    "does not exist");
          return;
          end;
/* Get the name and node of the process that should receive the message */
call find_owner_(procref, "process->op_con", op_conref);
process_name = extract_(procref, "process.name");
call find_owner_(tableref, "node->process", procref);
proc_node_name = extract_(tableref, "node_table.name");
/* If the process is not active, print an error message and return */
call find_owner_(ndm_proc_ownerref, "node/dbo/mg->process", procref);
if entity_class_name_(ndm_proc_ownerref) ^= "node_table"
     then do;
          call write_list_("Process", process_name, "at node",
                    proc_node_name, "is not active.  No message can be");
          call write_list_("          received over operator connection",
                    p_con_name);
          return;
          end;
/* Find out if the message can be received, or if the process must be
   blocked */
number_qd = extract_(op_conref, "op_con.number_qd");
if number_qd > 0
     then do;
          /* Remove one message from the queue */
          number_qd = number_qd - 1;
          call alter_(op_conref, "op_con.number_qd", number_qd);
          call write_list_(process_name, "at node", proc_node_name,
                    "has received a message");
          call write_list_("          over operator connection", p_con_name);
          return;
          end;
     else do;
          /* Block the process and initiate processing of an OBPL */
          call remove_(procref, "node/dbo/mg->process");
          call insert_(procref, "node/dbo/mg->process", "first",
                    op_conref);
          call write_list_("Process", process_name, "at node",
                    proc_node_name, "is blocked waiting for a");
          call write_list_("          message over operator connection",
                    p_con_name);
          call initiate_obpl(proc_node_name, process_name, proc_node_name,
                    p_con_name, "op_msg");
          return;
          end;
end OP_CON;
```

```
%:
RCV_CM: procedure:
          /* This procedure is a collection of subroutines which will accept
a control message and take the appropriate action.  The following user
visible function is included:
          RECEIVE CONTROL MESSAGE
The following support routines are included:
          PROCESS MESSAGE
          PROCESS OBPL PASS
          PROCESS "PROCESS TERMINATION"
          PROCESS RESOURCE GRANT
          PROCESS RESOURCE RELEASE
          PROCESS RESOURCE REQUEST      */


dcl       accept_node_name              char(12);
dcl       accept_node_tableref          fixed bin(17);
dcl       accept_proc_name              char(12);
dcl       accept_procref                fixed bin(17);
dcl       access_type                   char(9);
dcl       cont_msg_numb                 fixed bin;
dcl       cont_msgref                   fixed bin(17);
dcl       cont_msg_type                 char(20);
dcl       dbo_name                      char(12);
dcl       dbo_node_name                 char(12);
dcl       dbo_noderef                   fixed bin(17);
dcl       doref                         fixed bin(17);
dcl       dbo_tableref                  fixed bin(17);
dcl       eos                           bit(1);
dcl       mg_name                       char(12);
dcl       mgref                         fixed bin(17);
dcl       ndm_proc_ownerref             fixed bin(17);
dcl       obplref                       fixed bin(17);
dcl       p_cont_msg_numb               char(*);
dcl       p_msgref                      fixed bin(17);
dcl       p_obpl_passref                fixed bin(17);
dcl       p_res_grantref                fixed bin(17);
dcl       p_res_relref                  fixed bin(17);
dcl       p_res_reqref                  fixed bin(17);
dcl       process_name                  char(12);
dcl       proc_node_name                char(12);
dcl       proc_noderef                  fixed bin(17);
dcl       procref                       fixed bin(17);
dcl       proc_tableref                 fixed bin(17);
dcl       qd_msg_numb                   fixed bin;
dcl       rcv_msg_numb                  fixed bin;
dcl       rcv_node_name                 char(12);
dcl       sh_asmtref                    fixed bin(17);
dcl       write_list_                   entry options(variable);
%include DDM_serv_routines;
%include ADT_primitives;


/*        RECEIVE CONTROL MESSAGE                          6/15/76  */
receive_control_message:  rcvcm:  entry(p_cont_msg_numb);
/* This procedure will verify that the control message which has its number
   specified by "p_cont_msg_numb" has been sent, but has not been received.
   The procedure will then determine what type of control message it is, and
   the appropriate subroutine will be called to act on the message.  */
call find_first_(cont_msgref, "sys->control_message", SYS_REF, eos);
/* Convert the control message number from a character string to a numeric
   value */
cont_msg_numb = p_cont_msg_numb;
```

```
/* Find the control message with number specified by "p_cont_msg_numb" */
do while (^eos);
     if extract_(cont_msgref, "control_message.number") = cont_msg_numb
          then do:
               /* Remove the control message from the set of control messages
                  so that this control message will not be received a second
                  time */
               call remove_(cont_msgref, "sys->control_message");
               /* Find out what type of control message it is, and call the
                  routine that will take the appropriate action  */
               cont_msg_type = entity_class_name_(cont_msgref);
               if cont_msg_type = "msg"
                    then do;
                         call write_list_("Control message_number",
                                   p_cont_msg_numb, "representing a message",
                                   "in a message group");
                         call write_list_("          has been received");
                         call process_msg(cont_msgref);
                         return;
                         end;
               if cont_msg_type = "obpl_pass"
                    then do;
                         call write_list_("Control message number",
                                   p_cont_msg_numb, "representing an OBPL",
                                   "has been received.");
                         call process_obpl_pass(cont_msgref);
                         return;
                         end;
               if cont_msg_type = "res_grant"
                    then do;
                         call write_list_("Control message number",
                                   p_cont_msg_numb, "representing a remote",
                                   "resource allocation");
                         call write_list_("          has been received"):
                         call process_res_grant(cont_msgref);
                         return;
                         end;
               if cont_msg_type = "res_rel"
                    then do;
                         call write_list_("Control message number",
                                   p_cont_msg_numb, "representing a remote",
                                   "resource release"):
                         call write_list_("          has been received");
                         call process_res_rel(cont_msgref);
                         return;
                         end;
               if cont_msg_type = "res_req"
                    then do;
                         call write_list_("Control message number",
                                   p_cont_msg_numb, "representing a remote",
                                   "resource request");
                         call write_list_("          has been received");
                         call process_res_req(cont_msgref);
                         return:
                         end;
               end;
     call find_next_(cont_msgref, "sys->control_message", eos);
     end;
/* If "p_cont_msg_numb" didn't match any control message number, then we
   should print an error message and return  */
call write_list_(p_cont_msg_numb, " is not a valid control message number.",
        " Command rejected");
return:
```

```
/*         PROCESS MESSAGE                             7/1/76  */
process_msg:  entry(p_msgref);
/* This procedure will receive a message in a message group.  If a process
   is waiting for this message, it will be woken up, otherwise the message
   will be "queued" */
/* Get the name and location of the message group */
mg_name = extract_(p_msgref, "msg.mg_name");
eos = find_entity_loc(mgref, "sys->message", SYS_REF, mg_name,
      "message.name");
/* Acknowledge receipt of the message by adding 1 to the number of messages
   that have been queued in this message group */
qd_msg_numb = extract_(mgref, "message.number_qd") + 1;
call alter_(mgref, "message.number_qd", qd_msg_numb);
/* If no process has accepted the message group, return */
if ^inserted_(mgref, "rcv_proc->message")
     then do;
          call write_list_("Message group", mg_name, "has not been",
                    "accepted.  The message is queued.");
          return;
          end;
/* Get the name and node of the process that can receive the message */
call find_owner_(accept_procref, "rcv_proc->message", mgref);
accept_proc_name = extract_(accept_procref, "process.name");
call find_owner_(accept_node_tableref, "accept_node->message", mgref);
accept_node_name = extract_(accept_node_tableref, "node_table.name");
/* Keep the message queued if the process is not waiting for it.  Otherwise
   wakeup the process.  */
call find_owner_(ndm_proc_ownerref, "node/dbo/mg->process", accept_procref);
if ndm_proc_ownerref ^= mgref
     then call write_list_("No process is waiting for the message,",
               "so it is queued");
     else do;
          call remove_(accept_procref, "node/dbo/mg->process");
          call insert_(accept_procref, "node/dbo/mg->process", "first",
                    accept_node_tableref);
          rcv_msg_numb = extract_(mgref, "message.number_rcvd") + 1;
          call alter_(mgref, "message.number_rcvd", rcv_msg_numb);
          call write_list_("Process ", accept_proc_name, " at node ",
                    accept_node_name, "has been awakened upon");
          call write_list_("         receipt of a message in message group",
                    mg_name);
          end;
return;


/*         PROCESS OBPL PASS                           6/24/76  */
process_obpl_pass:  entry(p_obpl_passref);
/* This procedure will allow a partially expanded OBPL to be "received" by a
   node and then be expanded as much as possible within that node */
/* Get the location of the OBPL entity that has been "passed" between nodes.
   We need not check "eos" because we know the desired entity exists.  */
call find_first_(obplref, "obpl_pass->obpl", p_obpl_passref, eos);
/* Get the name of the node receiving the control message. */
rcv_node_name = extract_(p_obpl_passref, "obpl_pass.dest_node_name");
/* Remove the OBPL from this control message so that we can send the expanded
   OBPL in another control message if necessary */
call remove_(obplref, "obpl_pass->obpl");
/* Expand the OBPL as much as possible in the receiving node */
call exp_obpl(obplref, rcv_node_name);
return;
```

```
/*        PROCESS RESOURCE GRANT                    6/15/76  */
process_res_grant:  entry(p_res_grantref);
/* This procedure will wake up a process and give it access to a resource as
   specified by the remote resource grant control message pointed to by
   "p_res_grantref"  */
/* Get the names of the process, resource and nodes involved  */
process_name = extract_(p_res_grantref, "res_grant.proc_name");
proc_node_name = extract_(p_res_grantref, "res_grant.proc_node_name");
dbo_name = extract_(p_res_grantref, "res_grant.res_name");
dbo_node_name = extract_(p_res_grantref, "res_grant.res_node_name");
/* Find the locations of the entities for the process, resource and their node
   tables within the node specified by "proc_node_name".  Note that we need
   not test "eos" because we know the names placed in the control message
   represent existing entities.  */
eos = find_entity_loc(proc_noderef, "sys->node", SYS_REF, proc_node_name,
          "node.name");
eos = find_entity_loc(proc_tableref, "node->node_table", proc_noderef,
          proc_node_name, "node_table.name");
eos = find_entity_loc(procref, "node->process", proc_tableref, process_name,
          "process.name");
eos = find_entity_loc(dbo_tableref, "node->node_table", proc_noderef,
          dbo_node_name, "node_table.name");
eos = find_entity_loc(dboref, "node->dbo", dbo_tableref, dbo_name,
          "dbo.name");
/* Unblock the process  */
call remove_(procref, "node/dbo/mg->process");
call insert_(procref, "node/dbo/mg->process", "first", proc_tableref);
/* Give the process exclusive or shared access to the dbo, depending upon the
   type of access that was requested.  */
if extract_(procref, "process.access_type") = "exclusive"
    then do;
          /* Grant the process exclusive control of the database object */
          call insert_(dboref, "process->dbo", "first", procref);
          call write_list_(process_name, "at node", proc_node_name,
                    "has been granted exclusive use of");
          call write_list_("        ", dbo_name, "at node", dbo_node_name);
          return;
          end;
    else do;
          /* Grant the process shared access to the database object */
          call dcl_dbo_sh_asmt(sh_asmtref);
          call insert_(sh_asmtref, "dbo->dbo_sh_asmt", "first", dboref);
          call insert_(sh_asmtref, "process->dbo_sh_asmt", "first", procref);
          call write_list_(process_name, "at node", proc_node_name,
                    "has been granted shared access to");
          call write_list_("        ", dbo_name, "at node", dbo_node_name);
          return;
          end;
```

```
/*        PROCESS RESOURCE RELEASE                    6/15/76  */
process_res_rel: entry(p_res_relref);
/* This procedure will release a resource from control by a remote process,
   as specified in the resource release control message.  If possible,
   additional processes will be removed from the queue for the database
   object and will be granted access to the database object  */
/* Get the names of the process, resource and nodes involved */
process_name = extract_(p_res_relref, "res_rel.rel_proc_name");
proc_node_name = extract_(p_res_relref, "res_rel.rel_pnode_name");
dbo_name = extract_(p_res_relref, "res_rel.dest_dbo_name");
dbo_node_name = extract_(p_res_relref, "res_rel.dest_node_name");
/* Find the locations of the entities for the process, resource and their
   node tables within the node specified by "dbo_node_name".  Note that we
   do not test "eos" because we know the names placed in the resource release
   control message represent existing entities.  */
eos = find_entity_loc(dbo_noderef, "sys->node", SYS_REF, dbo_node_name,
        "node.name");
eos = find_entity_loc(dbo_tableref, "node->node_table", dbo_noderef,
        dbo_node_name, "node_table.name");
eos = find_entity_loc(dboref, "node->dbo", dbo_tableref, dbo_name,
        "dbo.name");
eos = find_entity_loc(proc_tableref, "node->node_table", dbo_noderef,
        proc_node_name, "node_table.name");
eos = find_entity_loc(procref, "node->process", proc_tableref,
        process_name, "process.name");
call write_list_(dbo_name, "at node", dbo_node_name, "has been released by");
call write_list_("        ", process_name, "at node", proc_node_name);
/* Check if the process had exclusive control of the database object */
if inserted_(dboref, "process->dbo")
     then do;
          /* Release the database object and then grant at least one other
             process access to the database object if any processes are
             queued for it */
          call remove_(dboref, "process->dbo");
          if ^empty_(dboref, "node/dbo/mg->process")
             then call rem_proc_from_queue(dboref, dbo_tableref);
          return;
          end;
     else do;
          /* Release the database object from this shared assignment, and if
             there are no other processes currently having shared access to
             the database object we can grant another process access to the
             database object if any are queued for it */
          call find_first_intersection_(sh_asmtref, "process->dbo_sh_asmt",
                    procref, "dbo->dbo_sh_asmt", dboref, eos);
          call delete_entity_(sh_asmtref, "dbo_sh_asmt");
          if member_count_(dboref, "dbo->dbo_sh_asmt") = 0
             then if ^empty_(dboref, "node/dbo/mg->process")
                  then call rem_proc_from_queue(dboref, dbo_tableref);
          return;
          end;
```

```
/*        PROCESS RESOURCE REQUEST                    6/15/76  */
process_res_req:  entry(p_res_reqref);
/* This procedure will process a request for a resource from a remote
   process, as specified in the resource request control message.  If
   the resource can be assigned, it will be and a control message will
   be generated to that effect.  Otherwise the process will remain blocked
   until the resource becomes available.  */
/* Get the names of the process, resource and nodes involved */
process_name = extract_(p_res_reqref, "res_req.req_proc_name");
proc_node_name = extract_(p_res_reqref, "res_req.req_node_name");
dbo_name = extract_(p_res_reqref, "res_req.dest_dbo_name");
dbo_node_name = extract_(p_res_reqref, "res_req.dest_node_name");
/* Find the locations of the entities for the process, resource and their
   respective node tables within the node specified by "dbo_node_name".  If
   the node is unaware of the existence of the process, create a local entity
   for that process.  e do not have to test eos because we know the entities
   for the node tables and the resource exist because the names were placed
   in the resource request control message  */
eos = find_entity_loc(dbo_noderef, "sys->node", SYS_REF, dbo_node_name,
          "node.name");
eos = find_entity_loc(dbo_tableref, "node->node_table", dbo_noderef,
          dbo_node_name, "node_table.name");
eos = find_entity_loc(dboref, "node->dbo", dbo_tableref, dbo_name,
          "dbo.name");
eos = find_entity_loc(proc_tableref, "node->node_table", dbo_noderef,
          proc_node_name, "node_table.name");
if find_entity_loc(procref, "node->process", proc_tableref, process_name,
          "process.name")
     then do;
          /* Create a "local" entity for the process, since one does not
             already exist */
          call dcl_process(procref, process_name);
          call insert_(procref, "node->process", "first", proc_tableref);
          call insert_(procref, "node/dbo/mg->process", "first",
                    proc_tableref);
          end;
/* Determine what type of access is desired */
access_type = extract_(p_res_reqref, "res_req.access_type");
/* Check if the database object might be available for assignment  */
if inserted_(dboref, "process->dbo") | ^empty_(dboref, "node/dbo/mg->process")
     then do;        /*Block the process if the database object has been
                       assigned to another process for exclusive use or
                       if other processes are currently queued for the
                       database object  */
          call alter_(procref, "process.access_type", access_type);
          call remove_(procref, "node/dbo/mg->process");
          call insert_(procref, "node/dbo/mg->process", "last", dboref);
          call write_list_("Resource not available, process remains blocked");
          call initiate_obpl(proc_node_name, process_name, dbo_node_name,
                    dbo_name, "dbo");
          return;
          end;
/* Check if the request is for shared access */
if access_type = "shared"
     then do;        /*Give the process shared access to the desired
                       database object  */
          call dcl_dbo_sh_asmt(sh_asmtref);
          call insert_(sh_asmtref, "dbo->dbo_sh_asmt", "first", dboref);
          call insert_(sh_asmtref, "process->dbo_sh_asmt", "first", procref);
          call write_list_(process_name, "at node", proc_node_name,
                    "is granted shared access to");
          call write_list_("        ", dbo_name, "at node", dbo_node_name);
          call dcl_rem_res_grant(dbo_node_name, dbo_name, proc_node_name,
                    process_name, cont_msg_numb);
          call write_list_("Control message number", cont_msg_numb,
                    "sent from", dbo_node_name, "to", proc_node_name);
          call write_list_("        representing this allocation");
```

```
          return;
          end;
/*The next if statement will be executed if the request is for
      exclusive use of the database object  */
/* Check if any process has shared access to the desired database object */
if ^ empty_(dboref, "dbo->dbo_sh_asmt")
     then do:          /*Queue the process for exclusive use of the database
                          object because at least one other process currently
                          has shared access to the database object.  */
          call alter_(procref, "process.access_type", "exclusive");
          call remove_(procref, "node/dbo/mg->process");
          call insert_(procref, "node/dbo/mg->process", "last", dboref);
          call write_list_("Resource is not currently available for",
                "exclusive use, process", process_name);
          call write_list_("         at node", proc_node_name,
                "remains blocked");
          call initiate_obpl(proc_node_name, process_name, dbo_node_name,
                dbo_name, "dbo");
          return;
          end;
     else do;          /*Grant the process exclusive use of the desired
                          database object.  */
          call insert_(dboref, "process->dbo", "first", procref);
          call write_list_(process_name, "at node", proc_node_name,
                "is granted exclusive use of");
          call write_list_("         ", dbo_name, "at node", dbo_node_name);
          call dcl_rem_res_grant(dbo_node_name, dbo_name, proc_node_name,
                process_name, cont_msg_numb);
          call write_list_("Control message number", cont_msg_numb,
                "sent from", dbo_node_name, "to", proc_node_name);
          call write_list_("         representing this allocation");
          return;
          end;
end RCV_CM;
```

```
%;
OBPL: procedure;
          /*  This procedure is a collection of subroutines which act on
an OBPL and check for deadlock.
The following support routines are included:
          CHECK FOR DEADLOCK
          COPY OBPL
          EXPAND OBPL
          OBPL ADD RESOURCE              */



dcl       eos                            bit(1);
dcl       first_procref                  fixed bin(17);
dcl       message_numb                   fixed bin;
dcl       mgref                          fixed bin(17);
dcl       ndm_proc_ownerref              fixed bin(17);
dcl       new_obplref                    fixed bin(17);
dcl       obpl_proc_name                 char(12);
dcl       obpl_proc_node_name            char(12);
dcl       obpl_proc_node_tableref        fixed bin(17);
dcl       obpl_procref                   fixed bin(17);
dcl       old_proc_entryref              fixed bin(17);
dcl       op_conref                      fixed bin(17);
dcl       operator_name                  char(12);
dcl       opref                          fixed bin(17);
dcl       p_copyref                      fixed bin(17);
dcl       p_eos                          bit(1);
dcl       p_ndm_proc_ownerref            fixed bin(17);
dcl       p_obplref                      fixed bin(17);
dcl       p_process_name                 char(12);
dcl       p_proc_node_name               char(12);
dcl       p_rcv_node_name                char(12);
dcl       proc_entryref                  fixed bin(17);
dcl       process_name                   char(12);
dcl       proc_node_name                 char(12);
dcl       procref                        fixed bin(17);
dcl       proc_tableref                  fixed bin(17);
dcl       rcv_noderef                    fixed bin(17);
dcl       res_name                       char(12);
dcl       res_node_name                  char(12);
dcl       res_node_tableref              fixed bin(17);
dcl       resref                         fixed bin(17);
dcl       res_type                       char(7);
dcl       sh_asmtref                     fixed bin(17);
dcl       write_list_                    entry options(variable);
%include DDM_serv_routines;
%include ADT_primitives;
```

```
/*        CHECK FOR DEADLOCK                            6/25/76  */
check_for_deadlock: entry(p_obplref, p_proc_node_name, p_process_name, p_eos);
/* This procedure will check if the process specified by "p_process_name"
   and located in the node specified by "p_proc_node_name" already has an
   entry in the OBPL pointed to by "p_obplref".  If no such entry exists,
   then one will be created and "p_eos" will be set to "1"b, indicating that
   there is no deadlock.  If an entry already exists for the process, we
   have a  deadlock and a message will be printed giving the processes
   involved and "p_eos" will be set to "0"b indicating a deadlock has been
   detected.  */
/* Get the location of the first proc_entry in the OBPL */
call find_first_(proc_entryref, "obpl->proc_entry", p_obplref, p_eos);
/* For each proc_entry in the OBPL, check if it matches the given process.
   Note that if we detect a deadlock, we will return from inside the loop
   and p_eos will be 0.  If no deadlock is detected we will exit the loop
   before returning and p_eos will be 1, as desired.  */
do while (^p_eos);
     /* If we have a match with "p_process_name" and a proc_entry, we must
        then check if the node name attribute matches "p_proc_node_name" */
     if p_process_name = extract_(proc_entryref, "proc_entry.process_name")
          then if p_proc_node_name = extract_(proc_entryref,
                    "proc_entry.node_name")
               then do;
                    /* A deadlock has been detected, list all the processes
                       involved and return.  */
                    call write_list_("A deadlock has been detected.",
                              "The following processes are involved:");
                    eos = "0"b;
                    do while (^eos);
                         process_name = extract_(proc_entryref,
                                   "proc_entry.process_name");
                         proc_node_name = extract_(proc_entryref,
                                   "proc_entry.node_name");
                         call write_list_("          ", process_name,
                                   "at node ", proc_node_name);
                         call find_prior_(proc_entryref, "obpl->proc_entry",
                                   eos);
                         end;
                    call write_list_("        End of deadlock list");
                    return;
                    end;
     /* Get the next proc_entry in the OBPL */
     call find_next_(proc_entryref, "obpl->proc_entry", p_eos);
     end;
/* No deadlock has been detected, so create a new proc_entry and have it
   inserted into the OBPL */
call dcl_proc_entry(p_obplref, p_proc_node_name, p_process_name);
return;
```

```
/*        COPY OBPL                                    6/25/76  */
copy_obpl:  entry(p_copyref, p_obplref);
/* This procedure will copy the OBPL pointed to by "p_obplref" and return
   a pointer to the copy via "p_copyref".  The order of the OBPL entries,
   and their attribute values in the copy will be identical to those in
   the original.  */
/* Get the attribute values (resource information) from the OBPL entity
   pointed to by "p_obplref".  */
res_name = extract_(p_obplref, "obpl.res_name");
res_node_name = extract_(p_obplref, "obpl.res_node_name");
res_type = extract_(p_obplref, "obpl.res_type");
/* Create an OBPL entity with the above attribute values */
call dcl_obpl(p_copyref, res_node_name, res_name, res_type);
message_numb = extract_(p_obplref, "obpl.msg_numb");
call alter_(p_copyref, "obpl.msg_numb", message_numb);
/* Get the last entry in the OBPL pointed to by "p_obplref"  */
call find_last_(old_proc_entryref, "obpl->proc_entry", p_obplref, eos);
/* Copy each OBPL entry */
do while (^eos);
     /* Get the attribute values of the proc_entry pointed to by
        "old_proc_entryref"  */
     process_name = extract_(old_proc_entryref, "proc_entry.process_name");
     proc_node_name = extract_(old_proc_entryref, "proc_entry.node_name");
     /* Create a new proc_entry with the above attribute values and
        insert it into the new copy of the OBPL.  */
     call dcl_proc_entry(p_copyref, proc_node_name, process_name);
     /* See if there are any more proc_entries to be copied  */
     call find_prior_(old_proc_entryref, "obpl->proc_entry", eos);
     end;
return;
```

```
/*          EXPAND OBPL                                6/24/76  */
exp_obpl:  entry(p_obplref, p_rcv_node_name);
/* This procedure will expand the OBPL pointed to by "p_obplref".  It will
   be expanded as much as poxxible using the information available to the
   node specified by "p_rcv_node_name"  */
/* Get the fully qualified name (resource name plus the name of the node
   in which it resides) of the resource which is controlled by or being
   waited for by the last process to be added to the OBPL.  (Note that we
   add processes to the OBPL by inserting them at the beginning of the set */
res_name = extract_(p_obplref, "obpl.res_name");
res_node_name = extract_(p_obplref, "obpl.res_node_name");
/* Get the type of the resource ("message" or "dbo" or "op_msg")  */
res_type = extract_(p_obplref, "obpl.res_type");
if res_type = "message"
    then do;
         /* The resource type is a message, therefore we know the process
            that can send the desired message is in the node that is
            expanding the OBPL.  We will act accordingly.  */
         /* Get the location of the entity for the message group from which
            a message is desired.  We need not test "eos" because we know
            the entity exists.  */
         eos = find_entity_loc(mgref, "sys->message", SYS_REF, res_name,
                   "message.name");
         /* Get the number (within the message group) of the message
            that is desired.  */
         message_numb = extract_(p_obplref, "obpl.msg_numb");
         /* If this number is less than or equal to the number of messages
            sent in this message group, then there is no deadlock.  */
         if  (message_numb > extract_(mgref, "message.number_sent"))
              then return;
         /* Find the process that can send the desired message */
         call find_owner_(procref, "send_proc->message", mgref);
         /* Find out if the process is active.  (If it is active there
            is no deadlock.)  */
         call find_owner_(ndm_proc_ownerref, "node/dbo/mg->process",
                   procref);
         if entity_class_name_(ndm_proc_ownerref) = "node_table"
              then return;
         /* Get the process name and check for deadlock */
         process_name = extract_(procref, "process.name");
         call check_for_deadlock(p_obplref, res_node_name, process_name,
                   eos);
         /* If eos = 0 then a deadlock has been detected and we are done */
         if (^eos)
              then return;
         /* Add the resource that the process is waiting for to the OBPL */
         call obpl_add_resource(p_obplref, ndm_proc_ownerref,
                   p_rcv_node_name, eos);
         /* If eos = 1 then the resource the process is waiting for is in
            the same node as the process, so we can continue to expand
            the OBPL.  */
         if eos
              then call exp_obpl(p_obplref, p_rcv_node_name);
         return;
         end;
if res_type = "op_msg"
    then do;
         /* The resource type is an operator message, therefore we know
            the last process to be added to the OBPL is waiting for a
            message from an operator at the same node.  We will act
            accordingly.  */
         /* Get the location of the entity for the operator connection
            over which a message is desired */
         eos = find_entity_loc(op_conref, "sys->op_con", SYS_REF,
                   res_name, "op_con.name");
         /* Get the location and name of the operator who can send the
            desired message */
```

```
          call find_owner_(opref, "operator->op_con", op_conref);
          operator_name = extract_(opref, "operator.name");
          /* Check if the operator is already in the OBPL list */
          call check_for_deadlock(p_obplref, res_node_name,
                    operator_name, eos);
          /* If eos = 0 then a deadlock has been detected and we are done */
          if (^eos)
             then return;
          /* Queue the OBPL and request status information from the
             operator */
          call insert_(p_obplref, "operator->obpl", "first", opref);
          call write_list_("An OBPL has been queued waiting for a status",
                    "report from operator", operator_name);
          call write_list_("          at node", res_node_name,
                    "The involved operator connection is", res_name);
          return;
          end;
/* If the next section is executed, a database object is controlled by or
   is being waited for by the last process to be added to the OBPL */
/* Get the name and location of the last process to be added to the OBPL */
call find_first_(obpl_procref, "obpl->proc_entry", p_obplref, eos);
obpl_proc_name = extract_(obpl_procref, "proc_entry.process_name");
obpl_proc_node_name = extract_(obpl_procref, "proc_entry.node_name");
/* Get the entity locations for the database object and its node table, and
   the process and its node table within the node specified by
   "p_rcv_node_name". We need not test "eos" in most cases because we
   know the entities exist  */
eos = find_entity_loc(rcv_noderef, "sys->node", SYS_REF, p_rcv_node_name,
          "node.name");
eos = find_entity_loc(obpl_proc_node_tableref, "node->node_table",
          rcv_noderef, obpl_proc_node_name, "node_table.name");
eos = find_entity_loc(res_node_tableref, "node->node_table", rcv_noderef,
          res_node_name, "node_table.name");
eos = find_entity_loc(obpl_procref, "node->process", obpl_proc_node_tableref,
          obpl_proc_name, "process.name");
/* We must test "eos" to see if the node containing the resource is aware
   of the existence of the most recently inserted process in the OBPL.  If
   it is not, we have no deadlock at this time, so we can return  */
if eos
    then return;
eos = find_entity_loc(resref, "node->dbo", res_node_tableref,
          res_name, "dbo.name");
/* Check if the resource is in the node that is expanding the OBPL */
if res_node_name = p_rcv_node_name
    then do;
          /* Verify that the process specified by "obpl_proc_name" is still
             waiting for the resource specified by "res_name" */
          call find_owner_(ndm_proc_ownerref, "node/dbo/mg->process",
                    obpl_procref);
          if resref ^= ndm_proc_ownerref
             then return;
          /* We must now add an entry to the OBPL for the process
             that controls the resource specified by "res_name",
             provided that the process is not already in the
             OBPL.  If there are n processes that have shared
             access to the database object, then we must create
             n copies of the OBPL and use a different copy
             for each reader  */
          if inserted_(resref, "process->dbo")
             then do;
                    /* The database object is held for exclusive
                       use.  Find the controlling process and
                       check for deadlock.  */
                    call find_owner_(procref, "process->dbo",
                              resref);
                    process_name = extract_(procref, "process.name");
                    call find_owner_(proc_tableref, "node->process",
```

```
                    procref);
            proc_node_name = extract_(proc_tableref,
                    "node_table.name");
            call find_owner_(ndm_proc_ownerref,
                    "node/dbo/mg->process", procref);
            /* If the process is active and it is at the
               same node as the resource, then we have
               no deadlock.  */
            if entity_class_name_(ndm_proc_ownerref)
                    = "node_table" & proc_node_name
                    = res_node_name
                then return;
            call check_for_deadlock(p_obplref,
                    proc_node_name, process_name, eos);
            /* If eos = 0, than a deadlock has been
               detected and we are done  */
            if (^eos)
                then return;
            if proc_node_name = res_node_name
                then do;
                    /* The process is in the same node as
                       the database object, so we can
                       continue to expand the OBPL  */
                    /* Add to the OBPL the resource that the process
                       is waiting for */
                    call obpl_add_resource(p_obplref,
                            ndm_proc_ownerref,
                            p_rcv_node_name, eos);
                    /* If eos = 1, then the resource that was added
                       to the OBPL is in the same node as the
                       process that is waiting for it, so we can
                       further expand the OBPL */
                    if eos
                        then call exp_obpl(p_obplref,
                                p_rcv_node_name);
                    return;
                    end;
                else do;
                    /* Send the OBPL to the node specified
                       by "proc_node_name"  */
                    call dcl_obpl_cont_msg(p_obplref,
                            proc_node_name, p_rcv_node_name);
                    return;
                    end;
            end;
    /* If the following code is executed, the database object
       has n readers.  We need to make n-1 additional copies
       of the OBPL.  Each time we make a copy of the OBPL,
       we expand that copy as much as possible for the given
       node and the process that we are associating with
       this copy  */
    /* Find a process that has shared access to the
       database object  */
    call find_first_(sh_asmtref, "dbo->dbo_sh_asmt",
            resref, eos);
    call find_owner_(first_procref, "process->dbo_sh_asmt",
            sh_asmtref);
    /* We will check for deadlock involving the OBPL and the
       process pointed to by "first_procref" after we check
       for deadlock with all the other readers of the
       database object.  We will tehererfore use the "original"
       OBPL (rather than a copy) for this check  */
    call find_next_(sh_asmtref, "dbo->dbo_sh_asmt", eos);
    do while (^eos);
         /* Find the process that has the shared access
            represented by the dbo_sh_asmt entity pointed
            to by "sh_asmtref"  */
```

```
        call find_owner_(procref, "process->dbo_sh_asmt",
                sh_asmtref);
        process_name = extract_(procref, "process.name");
        /* Get the name of the node in which the process
           resides */
        call find_owner_(proc_tableref, "node->process",
                procref);
        proc_node_name = extract_(proc_tableref,
                "node_table.name");
        /* If the process is not active or if it is at a node
           different from the node in which the resource
           resides, then we must check for deadlock.  */
        call find_owner_(ndm_proc_ownerref,
                "node/dbo/mg->process", procref);
        if entity_class_name_(ndm_proc_ownerref)
                ^= "node_table" | (proc_node_name
                ^= res_node_name)
            then do;
                /* Copy the OBPL and check for deadlock */
                call copy_obpl(new_obplref, p_obplref);
                call check_for_deadlock(new_obplref,
                        proc_node_name, process_name, eos);
                /* If eos = 1 then we must either continue
                   to expand the list or send it to
                   another node */
                if eos
                   then if proc_node_name = res_node_name
                        then do;
                            /* Add to the OBPL the resource that
                               the process is waiting for */
                            call obpl_add_resource(
                                    new_obplref,
                                    ndm_proc_ownerref,
                                    p_rcv_node_name, eos);
                            /* If eos = 1, then the resource that
                               was added to the OBPL is in the
                               same node as the process that is
                               waiting for it, so we can further
                               expand the OBPL */
                            if eos
                               then call exp_obpl(
                                        new_obplref,
                                        p_rcv_node_name);
                            end;
                        else call dcl_obpl_cont_msg(
                                new_obplref,
                                proc_node_name,
                                p_rcv_node_name);
                end;
        /* See if there are any more readers of the database
           object specified by "res_name" */
        call find_next_(sh_asmtref, "dbo->dbo_sh_asmt", eos);
        end;
/* Find the process name and the node in which it resides
   for the process pointed to by "first_procref" */
process_name = extract_(first_procref, "process.name");
call find_owner_(proc_tableref, "node->process",
        first_procref);
proc_node_name = extract_(proc_tableref, "node_table.name");
/* If the process is at the same node as the resource and
   it is active, we need not check for deadlock.  */
call find_owner_(ndm_proc_ownerref, "node/dbo/mg->process",
        first_procref);
if entity_class_name_(ndm_proc_ownerref) = "node_table"
        & proc_node_name = res_node_name
    then return;
/* Check for deadlock and then expand the OBPL or send
```

```
            it to another node */
        call check_for_deadlock(p_obplref, proc_node_name,
                process_name, eos);
        if eos
            then if proc_node_name = res_node_name
                then do;
                    call obpl_add_resource(p_obplref,
                            ndm_proc_ownerref, p_rcv_node_name,
                            eos);
                    if eos
                        then call exp_obpl(p_obplref,
                                p_rcv_node_name);
                    end;
                else call dcl_obpl_cont_msg(p_obplref,
                        proc_node_name, p_rcv_node_name);
        return;
        end;
/* The next section of code will be executed if the resource is located
   in a node different from the one that is expanding the list */
/* First check if the process is active.  If it is active, we are done */
call find_owner_(ndm_proc_ownerref, "node/dbo/mg->process", obpl_procref);
if entity_class_name_(ndm_proc_ownerref) = "node_table"
    then return;
/* Verify that the process specified by "obpl_proc_name" still controls
   the resource specified by "res_name" */
/* See if the process had either exclusive or shared access to
   the database object specified by "res_name".  If it has neither,
   we can return. */
if (empty_intersection_(obpl_procref, "process->dbo", res_node_tableref,
        "node->dbo")) & (empty_intersection_(obpl_procref,
        "process->dbo_sh_asmt", resref, "dbo->dbo_sh_asmt"))
    then return;
/* Add to the OBPL the resource that the process is waiting for */
call obpl_add_resource(p_obplref, ndm_proc_ownerref, p_rcv_node_name, eos);
/* If eos = 1, then the resource that was added to the OBPL is in the same
   node as the process that is waiting for it, so we can further expand
   the OBPL */
if eos
    then call exp_obpl(p_obplref, p_rcv_node_name);
return;
```

```
/*        OBPL ADD RESOURCE                             6/24/76  */
obpl_add_resource:  entry(p_obplref, p_ndm_proc_ownerref, p_rcv_node_name,
          p_eos);
/* This procedure will be passed a pointer to a resource that the most
   recently inserted process in an OBPL is waiting for.  The procedure will
   determine the type of resource that "p_ndm_proc_ownerref" points to, and
   will insert information about this resource into the OBPL entity pointed
   to by "p_obplref".  If the resource is in the node specified by
   "p_rcv_node_name", then p_eos will be set to 1, otherwise it will be set
   to 0 and the OBPL will be sent to the node that contains the resource */
if entity_class_name_(p_ndm_proc_ownerref) = "dbo"
    then do;
          /* Get the database object name and get the name of the node in
             which it resides */
          res_name = extract_(p_ndm_proc_ownerref, "dbo.name");
          call find_owner_(res_node_tableref, "node->dbo",
                    p_ndm_proc_ownerref);
          res_node_name = extract_(res_node_tableref, "node_table.name");
          call alter_(p_obplref, "obpl.res_type", "dbo");
          end;
if entity_class_name_(p_ndm_proc_ownerref) = "message"
    then do;
          /* Get the message group name and the name of the node from
             which a message should be coming */
          res_name = extract_(p_ndm_proc_ownerref, "message.name");
          call find_owner_(res_node_tableref, "init_node->message",
                    p_ndm_proc_ownerref);
          res_node_name = extract_(res_node_tableref, "node_table.name");
          /* Get the number of the message (within the message group) that
             is desired and insert this into the OBPL */
          message_numb = extract_(p_ndm_proc_ownerref, "message.number_qd")+1;
          call alter_(p_obplref, "obpl.msg_numb", message_numb);
          call alter_(p_obplref, "obpl.res_type", "message");
          end;
if entity_class_name_(p_ndm_proc_ownerref) = "op_con"
    then do;
          /* Get the name of the operator connection over which a message
             from an operator is desired */
          res_name = extract_(p_ndm_proc_ownerref, "op_con.name");
          /* The resource (operator connection) is located entirely in
             one node, so the resource_node_name is the same as that of
             the node processing the OBPL */
          res_node_name = p_rcv_node_name;
          call alter_(p_obplref, "obpl.res_type", "op_msg");
          end;
/* Put the resource name and its node name into the OBPL */
call alter_(p_obplref, "obpl.res_name", res_name);
call alter_(p_obplref, "obpl.res_node_name", res_node_name);
/* Check if the node can continue to expand the OBPL or if it must send the
   OBPL to another node */
if res_node_name = p_rcv_node_name
    then p_eos = "1"b;
    else do;
          p_eos = "0"b;
          call dcl_obpl_cont_msg(p_obplref, res_node_name, p_rcv_node_name);
return;
end OBPL;
```

```
%;
REL:  procedure;
        /*  This procedure contains subroutines which allow processes
to release resources and then assigns the released resource to a new
process if possible.  The following user visible function is included:
        RELEASE DATABASE OBJECT
The following support routine is included:
        REMOVE PROCESS FROM QUEUE    */


dcl     cont_msg_numb               fixed bin;
dcl     dbo_name                    char(12);
dcl     dbo_node_name               char(12);
dcl     dboref                      fixed bin(17);
dcl     dbo_tableref                fixed bin(17);
dcl     eos                         bit(1);
dcl     ndm_proc_ownerref           fixed bin(17);
dcl     ownerref                    fixed bin(17);
dcl     p_dbo_name                  char(*);
dcl     p_dbo_node_name             char(*);
dcl     p_dboref                    fixed bin(17);
dcl     p_dbo_tableref              fixed bin(17);
dcl     pnoderef                    fixed bin(17);
dcl     p_process_name              char(*);
dcl     p_proc_node_name            char(*);
dcl     process_name                char(12);
dcl     proc_node_name              char(12);
dcl     procref                     fixed bin(17);
dcl     ptableref                   fixed bin(17);
dcl     res_rel_ref                 fixed bin(17);
dcl     sec_node_name               char(12);
dcl     sh_asmtref                  fixed bin(17);
dcl     tableref                    fixed bin(17);
dcl     temp_name                   char(12);
dcl     write_list_                 entry options(variable);
%include DDM_serv_routines;
%include ADT_primitives;
```

```
/*        RELEASE DATABASE OBJECT                      6/2/76  */
release_dbo:  rldbo:  entry(p_proc_node_name, p_process_name, p_dbo_node_name,
          p_dbo_name);
/* This procedure will cause the process specified by "p_process_name" (at
   node "p_proc_node_name") to release its control over the database object
   specified by "p_dbo_name" and located at the node specified by
   "p_dbo_node_name" */
/* Verify that the node specified by "p_proc_node_name" exists  */
if find_entity_loc(pnoderef, "sys->node", SYS_REF, p_proc_node_name,
          "node.name");
     then do;
          call write_list_("Invalid process node name.  ", p_proc_node_name,
                    "does not exist");
          return;
          end;
/* Verify that the process specified by "p_process_name" exists  at the node
   specified by "p_proc_node_name"  */
eos = find_entity_loc(ptableref, "node->node_table", pnoderef,
          p_proc_node_name, "node_table.name");
if find_entity_loc(procref, "node->process", ptableref, p_process_name,
          "process.name")
     then do;
          call write_list_("Invalid process name.", p_process_name, "at node",
                    p_proc_node_name, "does not exist");
          return;
          end;
/* Verify that the node specified by "p_dbo_node_name" exists */
if find_entity_loc(dbo_tableref, "node->node_table", pnoderef,
          p_dbo_node_name, "node_table.name")
     then do;
          call write_list_("Invalid database object node name.  ",
                    p_dbo_node_name, "does not exist.");
          return;
          end;
/* Verify that the  database object specified by "p_dbo_name" exists at the
   node specified by "p_dbo_node_name" and that the process specified by
   "p_process_name" has access to  it.  */
/* Verify that the node containing the process is aware of the existence of
   the database object */
if find_entity_loc(dboref, "node->dbo", dbo_tableref, p_dbo_name, "dbo.name")
     then do;
          call write_list_("Invalid release.  Process", p_process_name,
                    "at node", p_proc_node_name, "does not have");
          call write_list_("          access to", p_dbo_name, "at node",
                    p_dbo_node_name);
          return;
          end;
/* Verify that the process has access to the database object */
if find_entity_loc(dboref, "process->dbo", procref, p_dbo_name, "dbo.name")
          & empty_intersection_(procref, "process->dbo_sh_asmt", dboref,
          "dbo->dbo_sh_asmt")
     then do;
          call write_list_("Invalid release.  Process", p_process_name,
                    "at node", p_proc_node_name, "does not have");
          call write_list_("          access to", p_dbo_name, "at node",
                    p_dbo_node_name);
          return;
          end;
/* Verify that the process is active */
call find_owner_(ndm_proc_ownerref, "node/dbo/mg->process", procref);
if entity_class_name_(ndm_proc_ownerref) ^= "node_table"
     then do;
          call write_list_("Invalid release.  Process ", p_process_name,
                    "at node", p_proc_node_name, "is not active");
          return;
          end;
/* Check if the database object is at a node different from the one that
```

```
    contains the process */
if p_proc_node_name ^= p_dbo_node_name
   then do;
        /* Release the resource and send a resource release control message
           to the node which contains the database object */
        /* Check  if there are no more "local" processes queued for  the
           specified remote database object */
        if empty_(dboref, "node/dbo/mg->process")
           then do;
                /* If the process had exclusive control of the  database
                   object or if no other local process had shared access
                   to the database object, then we can delete all local
                   information about the remote database object,
                   otherwise just "release" the shared access of the
                   process to the database object */
                if inserted_(dboref, "process->dbo")
                      | member_count_(dboref, "dbo->dbo_sh_asmt") < 2
                   then call delete_entity_(dboref, "dbo");
                   else do;
                        /* Find the entity for the involved dbo_sh_asmt
                           and delete it */
                        call find_first_intersection_(sh_asmtref,
                                    "process->dbo_sh_asmt", procref,
                                    "dbo->dbo_sh_asmt", dboref, eos);
                        call delete_entity_(sh_asmtref, "dbo_sh_asmt");
                        end;
                end;
           else do;
                /* Release the database object from access by the process,
                   but retain other local information about the remote
                   database object */
                if inserted_(dboref, "process->dbo")
                   then call remove_(dboref, "process->dbo");
                   else do;
                        call find_first_intersection_(sh_asmtref,
                                    "process->dbo_sh_asmt", procref,
                                    "dbo->dbo_sh_asmt", dboref, eos);
                        call  delete_entity_(sh_asmtref, "dbo_sh_asmt");
                        end;
                end;
        /* Create an entity for a remote resource release and the declare it
           as a control message */
        call create_entity_(res_rel_ref, "res_rel");
        call create_attribute_(res_rel_ref, "res_rel.rel_pnode_name",
                "field, 12, p_proc_node_name);
        call create_attribute_(res_rel_ref, "res_rel.rel_proc_name",
                "field", 12, p_process_name);
        call create_attribute_(res_rel_ref, "res_rel.dest_node_name",
                "field", 12, p_dbo_node_name);
        call create_attribute_(res_rel_ref, "res_rel.dest_dbo_name",
                "field", 12, p_dbo_name);
        call dcl_control_message(res_rel_ref, "res_rel", cont_msg_numb);
        call insert_(res_rel_ref, "sys->control_message",  "last", SYS_REF);
        call write_list_("Control message number", cont_msg_numb,
                "sent from", p_proc_node_name, "to", p_dbo_node_name);
        call write_list_("         representing a remote resource release");
        return;
        end;
/* The next section will be executed if the process and database object are
   located in the same node */
/* Check if the process had exclusive control of the database object */
if inserted_(dboref, "process->dbo")
   then do;
        /* Release the database object and then grant at least one other
           process access to the database object if any processes are
           queued for it. */
        call remove_(dboref, "process->dbo");
```

```
        call write_list_("Process", p_process_name, "at node",
                p_proc_node_name, "has released database");
        call write_list_("          object", p_dbo_name, "at node",
                p_dbo_node_name);
        if ^empty_(dboref, "node/dbo/mg->process")
            then call rem_proc_from_queue(dboref, dbo_tableref);
        return;
        end;
    else do;
        /* Release the database object from this shared assignment, and if
           there are no other processes currently having shared access to
           the database object we can grant another process access to the
           database object if any are queued for it */
        call find_first_intersection_(sh_asmtref, "process->dbo_sh_asmt",
                procref, "dbo->dbo_sh_asmt", dboref, eos);
        call delete_entity_(sh_asmtref, "dbo_sh_asmt");
        call write_list_("Process", p_process_name, "at node",
                p_proc_node_name, "has released database");
        call write_list_("          object", p_dbo_name, "at node",
                p_dbo_node_name);
        if member_count_(dboref, "dbo->dbo_sh_asmt") = 0
            then if ^empty_(dboref, "node/dbo/mg->process")
                then call rem_proc_from_queue(dboref, dbo_tableref);
        return;
        end;


/*          REMOVE PROCESS FROM QUEUE                  6/3/76  */
rem_proc_from_queue:  entry(p_dboref, p_dbo_tableref);
/* This procedure will grant at least one process access to the database
   object that is referenced by "p_dboref" (and is located in the node that
   has its own node_table referenced by "p_dbo_tableref").  If the first
   process on the queue wants exclusive use of the database object, then
   only this process will be granted access to the dbo, otherwise all
   processes that requested shared access and that are in front (in the
   queue) of all processes that requested exclusive use will be given
   shared access to the database object */
/* Find the first process queued for the database object */
call find_first_(procref, "node/dbo/mg->process", p_dboref, eos);
/* Check if the process wants exclusive use of the database object */
if extract_(procref, "process.access_type") = "exclusive"
    then do;
        /* Unblock the process */
        call find_owner_(ownerref, "node->process", procref);
        call remove_(procref, "node/dbo/mg->process");
        call insert_(procref, "node/dbo/mg->process", "first", ownerref);
        /* Give the process exclusive control of the resource */
        call insert_(p_dboref, "process->dbo", "first", procref);
        /* Get the names of the process, dbo and nodes involved */
        dbo_name = extract_(p_dboref, "dbo.name");
        dbo_node_name = extract_(p_dbo_tableref, "node_table.name");
        process_name = extract_(procref, "process.name");
        proc_node_name = extract_(ownerref, "node_table.name");
        /* Check if the process gaining access to the database object is in
           the same node as the dbo */
        if proc_node_name = dbo_node_name
            then do;
                call write_list_("Process", process_name, "at node",
                        proc_node_name, "is given exclusive use of");
                call write_list_("        ", dbo_name, "at node",
                        dbo_node_name);
                return;
                end;
            else do;
                /* Create a control message for a remote resourc
```

```
                    allocation and send it across nodes.  */
                 call dcl_rem_res_grant(dbo_node_name, dbo_name,
                        proc_node_name, process_name, cont_msg_numb);
                 call write_list_("Control message number", cont_msg_numb,
                        "sent from", dbo_node_name, "to",
                        proc_node_name);
                 call write_list_("          granting", process_name,
                        "exclusive use of", dbo_name);
                 return;
                 end;
           end;
     else do while (^eos);
           /* The first process on the queue requested shared access */
           /* Unblock the process */
           call find_owner_(ownerref, "node->process", procref);
           call remove_(procref, "node/dbo/mg->process");
           call insert_(procref, "node/dbo/mg->process", "first", ownerref);
           /* Give the process shared access to the database object */
           call dcl_dbo_sh_asmt(sh_asmtref);
           call insert_(sh_asmtref, "dbo->dbo_sh_asmt", "first", p_dboref);
           call insert_(sh_asmtref, "process->dbo_sh_asmt", "first", procref);
           /* Get the names of the process, dbo and nodes involved */
           dbo_name = extract_(p_dboref, "dbo.name");
           dbo_node_name = extract_(p_dbo_tableref, "node_table.name");
           process_name = extract_(procref, "process.name");
           proc_node_name = extract_(ownerref, "node_table.name");
           /* Check if the process gaining access to the database object is in
              the same node as the dbo */
           if proc_node_name = dbo_node_name
              then do;
                 call write_list_("Process", process_name, "at node",
                        proc_node_name, "is granted shared access to");
                 call write_list_("          ", dbo_name, "at node",
                        dbo_node_name);
                 end;
              else do;
                 /* Create a control message for a remote resource
                    allocation and send it across nodes.  */
                 call dcl_rem_res_grant(dbo_node_name, dbo_name,
                        proc_node_name, process_name, cont_msg_numb);
                 call write_list_("Control message number", cont_msg_numb,
                        "sent from", dbo_node_name, "to",
                        proc_node_name);
                 call write_list_("          granting", process_name,
                        "shared access to", dbo_name);
                 end;
           /* Find what is now the first process queued for the database
              object */
           call find_first_(procref, "node/dbo/mg->process", p_dboref, eos);
           /* If this process wants exclusive control of the database object it
              must remain blocked and we will not remove any more processes
              from the queue */
           if extract_(procref, "process.access_type") = "exclusive"
              then eos = "1"b;
           end;
return;
end REL;
```

```
/*          DDM_serv_routines.incl.pl1
The following declarations are of the DDM service routines  */
dcl         check_for_deadlock              entry(fixed bin(17), char(12),
                                                  char(12), bit(1));
                        /* Located within Procedure OBPL */
dcl         dcl_dbo                         entry(fixed bin(17), char(12));
                        /* Located within Procedure DDM */
dcl         dcl_dbo_sh_agmt                 entry(fixed bin(17));
                        /* Located within Procedure DDM */
dcl         dcl_control_message             entry(fixed bin(17), char(20),
                                                  fixed bin);
                        /* Located within Procedure DDM */
dcl         dcl_node_table                  entry(fixed bin(17), char(12));
                        /* Located within Procedure DDM */
dcl         dcl_obpl                        entry(fixed bin(17), char(12),
                                                  char(12), char(7));
                        /* Located within Procedure DDM */
dcl         dcl_obpl_cont_msg               entry(fixed bin(17), char(12),
                                                  char(12));
                        /* Located within Procedure DDM */
dcl         dcl_proc_entry                  entry(fixed bin(17), char(12),
                                                  char(*));
                        /* Located within Procedure DDM */
dcl         dcl_process                     entry(fixed bin(17), char(12));
                        /* Located within Procedure DDM */
dcl         dcl_proc_term                   entry(char(12), char(*), char(12));
                        /* Located within Procedure RCV_CM */
dcl         dcl_rem_res_grant               entry(char(12), char(*), char(12),
                                                  char(*), fixed bin);
                        /* Located within Procedure DDM */
dcl         find_entity_loc                 entry(fixed bin(17), char(20),
                                                  fixed bin(17), char(12),
                                                  char(44)) returns (bit(1));
                        /* Located within Procedure DDM */
dcl         exp_obpl                        entry(fixed bin(17), char(12));
                        /* Located within Procedure OBPL */
dcl         initiate_obpl                   entry(char(12), char(*), char(12),
                                                  char(12), char(7));
                        /* Located within Procedure DDM */
dcl         obpl_add_resource               entry(fixed bin(17), fixed bin(17),
                                                  char(12), bit(1));
                        /* Located within Procedure OBPL */
dcl         rem_proc_from_queue             entry(fixed bin(17), fixed bin(17));
                        /* Located within Procedure REL */
dcl         rldbo                           entry(char(*), char(*), char(*),
                                                  char(*));
                        /* Located within Procedure REL */
```

```
/*75-12-29                ADT_primitives.incl.pl1
These are ADT primitives designed to assist the function definition writer */
dcl     add_                          entry(char(128), char(128))
                                          returns(char(128) varying);
dcl     alter_                        entry(fixed bin(17), char(44),
                                          char(*));
dcl     append_                       entry(fixed bin(17), char(20),
                                          char(6), fixed bin(17));
dcl     break_                        entry(fixed bin(17));
dcl     create_attribute_             entry(fixed bin(17), char(44),
                                          char(10), fixed bin(17),
                                          char(*));
dcl     create_catalog_object_        entry(fixed bin(17), char(*));
dcl     create_entity_                entry(fixed bin(17), char(20));
dcl     create_group_                 entry(fixed bin(17), char(44));
dcl     create_order_                 entry(fixed bin(17), char(20),
                                          char(20));
dcl     create_relationship_          entry(fixed bin(17), char(20),
                                          char(9));
dcl     deleted_                      entry(fixed bin(17)) returns(bit(1));
dcl     delete_entity_                entry(fixed bin(17), char(20));
dcl     divide_                       entry(char(128), char(128))
                                          returns(char(128) varying):
dcl     empty_                        entry(fixed bin(17), char(20))
                                          returns(bit(1));
dcl     empty_intersection_           entry(fixed bin(17), char(20),
                                          fixed bin(17), char(20))
                                          returns(bit(1));
dcl     entity_class_name_            entry(fixed bin(17))
                                          returns(char(20));
dcl     entity_order_name_            entry(fixed bin(17), char(20))
                                          returns(bit(1));
dcl     exception_                    entry;
dcl     extract_                      entry(fixed bin(17), char(44))
                                          returns(char(128) varying);
dcl     find_associatively_           entry(fixed bin(17), char(20),
                                          fixed bin(17), char(128) varying,
                                          char(44), bit(1));
dcl     find_catalog_object_          entry(fixed bin(17), char(*));
dcl     find_direct_                  entry(fixed bin(17), char(*));
dcl     find_each_                    entry(fixed bin(17), bit(1));
dcl     find_first_                   entry(fixed bin(17), char(20),
                                          fixed bin(17), bit(1));
dcl     find_first_intersection_      entry(fixed bin(17), char(20),
                                          fixed bin(17), char(20),
                                          fixed bin(17), bit(1));
dcl     find_first_union_             entry(fixed bin(17), char(20),
                                          fixed bin(17), char(20),
                                          fixed bin(17), bit(1));
dcl     find_last_                    entry(fixed bin(17), char(20),
                                          fixed bin(17), bit(1));
dcl     find_next_                    entry(fixed bin(17), char(20),
                                          bit(1));
dcl     find_next_intersection_       entry(fixed bin(17), char(20),
                                          char(20), bit(1));
dcl     find_next_union_              entry(fixed bin(17), char(20),
                                          fixed bin(17), char(20),
                                          fixed bin(17), bit(1));
dcl     find_owner_                   entry(fixed bin(17), char(20),
                                          fixed bin(17));
dcl     find_prior_                   entry(fixed bin(17), char(20),
                                          bit(1));
dcl     insert_                       entry(fixed bin(17), char(20),
                                          char(6), fixed bin(17));
dcl     inserted_                     entry(fixed bin(17), char(20))
                                          returns(bit(1));
```

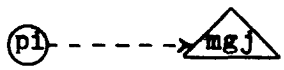
```
dcl      last_of_set_            entry(fixed bin(17), char(20))
                                      returns(bit(1));
dcl      member_                 entry(fixed bin(17), char(20))
                                      returns(bit(1));
dcl      member_count_           entry(fixed bin(17), char(20))
                                      returns(fixed bin(17));
dcl      name_catalog_object_    entry(fixed bin(17))
                                      returns(char(44) varying);
dcl      multiply_               entry(char(128), char(128))
                                      returns(char(128) varying);
dcl      owner_                  entry(fixed bin(17), char(20))
                                      returns(bit(1));
dcl      remove_                 entry(fixed bin(17), char(20));
dcl      sort_relationship_      entry(fixed bin(17), char(20),
                                      char(20));
dcl      subtract_               entry(char(128), char(128))
                                      returns(char(128) varying);
/* The following are global reference variables used by modellers */
dcl      changemode              fixed bin(17) external static;
dcl      SF_REF                  fixed bin(17) external static init(0);
dcl      CN_REF                  fixed bin(17) external static init(0);
dcl      PROC_REF                fixed bin(17) external static init(0);
dcl      PSPH_REF                fixed bin(17) external static init(0);
dcl      PSSG_REF                fixed bin(17) external static init(0);
dcl      return_code             fixed binary external static init(0);
dcl      SYS_REF                 fixed bin(17) external static init(0);
dcl      tracemode               bit(1)  external static init("0"b);
```

Appendix III

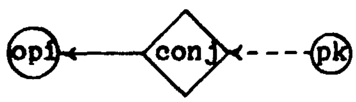
This appendix contains examples of several deadlock and "near deadlock" situations, thus demonstrating various features of the deadlock detection algorithm presented in Chapter VI. In the case where a deadlock is detected, a final state diagram is given, whereas in the examples where no deadlock is detected, an important intermediate state is also shown. A key to the diagrams appears on the next page. Diagrams appear on a page with a header containing the name(s) of the associated scenario(s). Each diagram immediately follows the first scenario with which it is associated.

It should be noted that before the commands specific to each example were executed, after the system state was reinitialized, the commands in file "demo0" were executed.

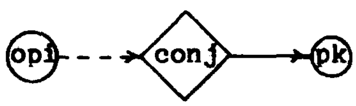
Represents process "pi" as the initiator of message group "mgj". pi and mgj are always in the same node for this representation.



Represents process "pi" as the acceptor of message group "mgj" and "pi" is currently waiting for a message in "mgj". pi and mgj need not be in the same node for this representation.

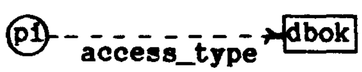


Represents process "pk" waiting for a message from operator "opi" over operator connection "conj".

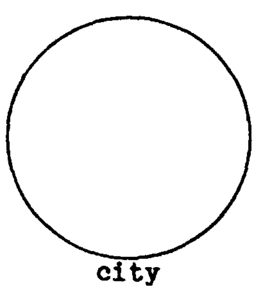


Represents operator "opi" waiting for a message from process "pk" over operator connection "conj".

Represents process "pi" as having access to database object "dbok". The type of access is specified by "access_type". pi and $\boxed{\text{dbok}}$ need not be in the same node.



Represents process "pi" as waiting for access to database object "dbok". The type of access desired is specified by "access_type". pi and $\boxed{\text{dbok}}$ need not be in the same node.

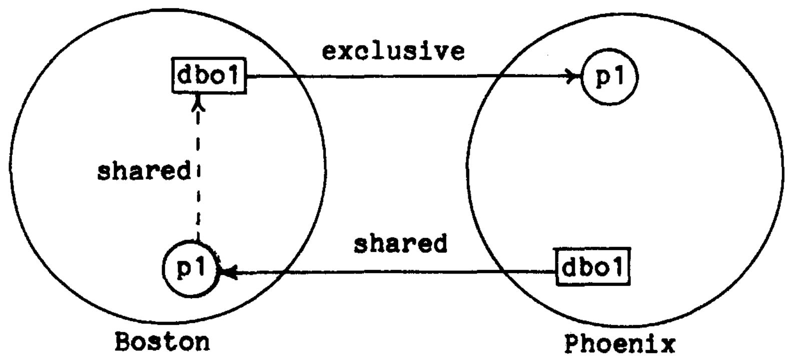


Represents a node with the node name specified by "city". pi and $\boxed{\text{dbok}}$ drawn "within" this node represent processes and database objects located within the node specified by "city". mgj drawn "within" the node represents a message group that was initiated by a process located in the node specified by "city".

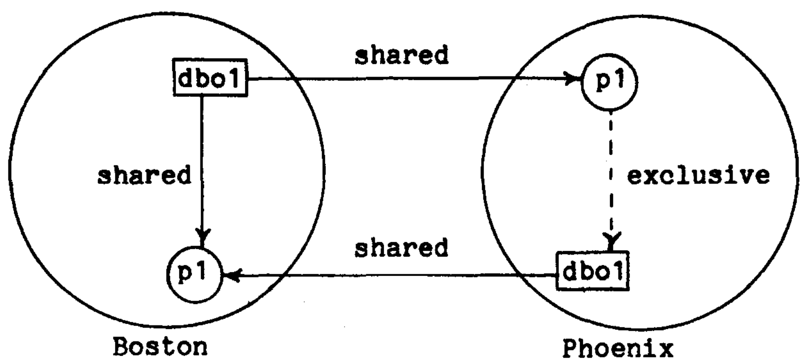
```
scenario demo0
sysgen
  System created
cnode Boston
  Node created:    Boston
cnode Phoenix
  Node created:    Phoenix
cproc Boston p1
  Process     p1  created in node    Boston
cproc Boston p2
  Process     p2  created in node    Boston
cproc Boston p3
  Process     p3  created in node    Boston
cdbo Boston dbo1
  Database object    dbo1    created in node    Boston
cdbo Boston dbo2
  Database object    dbo2    created in node    Boston
cproc Phoenix p1
  Process     p1  created in node    Phoenix
cproc Phoenix p2
  Process     p2  created in node    Phoenix
cproc Phoenix p3
  Process     p3  created in node    Phoenix
cdbo Phoenix dbo1
  Database object    dbo1    created in node    Phoenix
cdbo Phoenix dbo2
  Database object    dbo2    created in node    Phoenix
cnode Cambridge
  Node created:    Cambridge
cproc Cambridge p1
  Process     p1  created in node    Cambridge
cproc Cambridge p2
  Process     p2  created in node    Cambridge
cproc Cambridge p3
  Process     p3  created in node    Cambridge
cdbo Cambridge dbo1
  Database object    dbo1    created in node    Cambridge
cdbo Cambridge dbo2
  Database object    dbo2    created in node    Cambridge
```

```
scenario demo_bug
note     This is an example of a case where a deadlock involving two
note     processes and two resources located in two nodes is detected,
note     when in fact no deadlock exists.  The reason a deadlock is
note     detected is that an OBPL sent from Boston to Phoenix had its
note     arrival delayed long enough so that p1 in Phoenix could release
note     dbo1 in Boston, request access to it again, gain use of the
note     database object and then request access to and get queued for
note     dbo1 in Phoenix before Phoenix examined the OBPL.  The first
note     seven commands set up the state where p1 in Phoenix has exclusive
note     use of dbo1 in Boston, p1 in Boston has shared use of dbo1 in
note     Phoenix, p1 in Boston is blocked waiting for shared use of dbo1
note     in Boston, and an OBPL has been sent to Phoenix by Boston.
rqdbo shared Boston p1 Phoenix dbo1
  Process  p1  at node  Boston  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  1  sent from  Boston  to  Phoenix
           representing a remote resource request
rcvcm 1
  Control message number  1  representing a remote  resource request
           has been received
  p1             at node  Boston        is granted shared access to
           dbo1          at node  Phoenix
  Control message number  2  sent from  Phoenix       to  Boston
           representing this allocation
rcvcm 2
  Control message number  2  representing a remote  resource allocation
           has been received
  p1             at node  Boston        has been granted shared access to
           dbo1          at node  Phoenix
rqdbo exclusive Phoenix p1 Boston dbo1
  Process  p1  at node  Phoenix  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  3  sent from  Phoenix  to  Boston
           representing a remote resource request
rcvcm 3
  Control message number  3  representing a remote  resource request
           has been received
  p1             at node  Phoenix       is granted exclusive use of
           dbo1          at node  Boston
  Control message number  4  sent from  Boston        to  Phoenix
           representing this allocation
rcvcm 4
  Control message number  4  representing a remote  resource allocation
           has been received
  p1             at node  Phoenix       has been granted exclusive use of
           dbo1          at node  Boston
rqdbo shared Boston p1 Boston dbo1
  Resource not available, process blocked.
  Control message number  5  sent from  Boston        to  Phoenix
           representing an OBPL
note     Do not let the OBPL be received at this time.  Let p1 in Phoenix
note     release dbo1 in Boston, so that p1 in Boston will be awakened and
note     granted shared use of dbo1 in Boston.
rldbo Phoenix p1 Boston dbo1
  Control message number  6  sent from  Phoenix  to  Boston
           representing a remote resource release
rcvcm 6
  Control message number  6  representing a remote  resource release
           has been received
  dbo1           at node  Boston        has been released by
           p1            at node  Phoenix
  Process  p1            at node  Boston        is granted shared access to
           dbo1          at node  Boston
note     Let p1 in Phoenix request access to dbo1 in Boston for the
note     second time, and let it be granted shared use of the database
note     object.
```

```
rqdbo shared Phoenix p1 Boston dbo1
  Process  p1  at node  Phoenix  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  7  sent from  Phoenix  to  Boston
           representing a remote resource request
rcvcm 7
  Control message number  7  representing a remote  resource request
           has been received
  p1          at node  Phoenix       is granted shared access to
           dbo1          at node  Boston
  Control message number  8  sent from  Boston         to  Phoenix
           representing this allocation
rcvcm 8
  Control message number  8  representing a remote  resource allocation
           has been received
  p1          at node  Phoenix       has been granted shared access to
           dbo1          at node  Boston
note     Let p1 in Phoenix request exclusive use of dbo1 in Phoenix.
note     The process will be blocked and an OBPL will be sent to Boston
note     where it will be discarded because p1 in Boston is active.
rqdbo exclusive Phoenix p1 Phoenix dbo1
  Resource is not currently available for  exclusive use, process  p1
           at node  Phoenix  is blocked.
  Control message number  9  sent from  Phoenix        to  Boston
           representing an OBPL
rcvcm 9
  Control message number  9  representing an OBPL  has been received.
note     Now let Phoenix receive the OBPL that was previously sent by
note     Boston.  A "false" deadlock will be detected because p1 in Phoenix
note     is blocked and has access to dbo1 in Boston, even though this is
note     not the same assignment of the resource that was used when the
note     OBPL was created.
rcvcm 5
  Control message number  5  representing an OBPL  has been received.
  A deadlock has been detected.  The following processes are involved:
                    p1              at node   Boston
                    p1              at node   Phoenix
           End of deadlock list
```

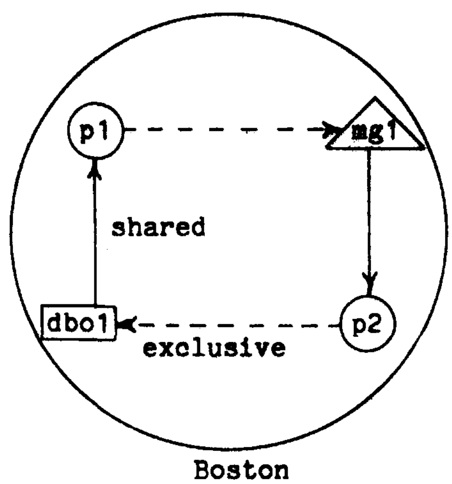


State where control message 5 representing an OBPL has just been sent from Boston to Phoenix. Receipt of the OBPL is delayed until after the state drawn below has drawn been reached.



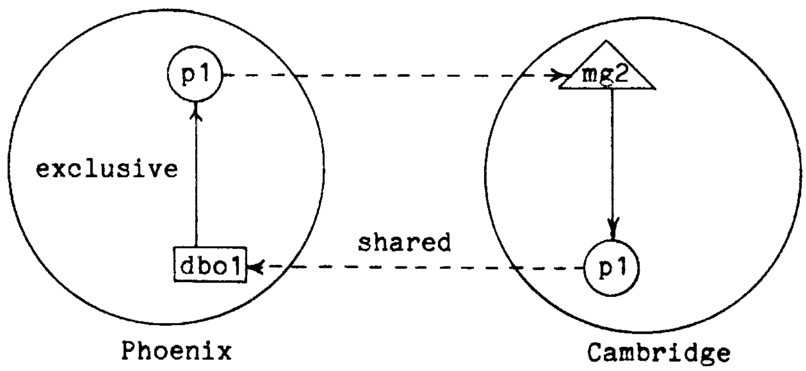
Final State Diagram

```
scenario demo1
note     This is an example of a two process two resource deadlock in a
note     single node.  No control messages and no operators are involved
note     in the detection of this deadlock.
initmg mg1 Boston p2 Boston
  Message group   mg1  has been initiated
acceptmg mg1 Boston p1
  mg1   has been accepted by   p1   at node   Boston
rqdbo shared Boston p1 Boston dbo1
  p1  at node  Boston  granted shared access to  dbo1  at node  Boston
rqdbo exclusive Boston p2 Boston dbo1
  Resource is not currently available for  exclusive use, process  p2
           at node  Boston  is blocked.
rcvmsg mg1
  Process   p1            at node  Boston        is blocked waiting for a
           message in message group  mg1
  A deadlock has been detected.  The following processes are involved:
                    p1            at node   Boston
                    p2            at node   Boston
           End of deadlock list
```



Final State Diagram

```
scenario demo2
note     This is an example of a two process two resource deadlock
note     involving two nodes.  The first three commands create the state
note     where both processes are active and both involved resources have
note     been allocated to the proper processes.
rqdbo exclusive Phoenix p1 Phoenix dbo1
  p1  at node  Phoenix  is granted exclusive use of  dbo1  at node  Phoenix
initmg mg2 Cambridge p1 Phoenix
  Message group   mg2  has been initiated
acceptmg mg2 Phoenix p1
  mg2   has been accepted by   p1   at node   Phoenix
rqdbo shared Cambridge p1 Phoenix dbo1
  Process  p1  at node  Cambridge  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  1  sent from  Cambridge  to  Phoenix
           representing a remote resource request
note     We will delay the receipt by Phoenix of this resource request.
rcvmsg mg2
  Process   p1             at node  Phoenix       is blocked waiting for a
           message in message group  mg2
  Control message number  2  sent from  Phoenix       to  Cambridge
           representing an OBPL
rcvcm 2
  Control message number  2  representing an OBPL  has been received.
  Control message number  3  sent from  Cambridge     to  Phoenix
           representing an OBPL
note     This OBPL contains entries for p1 in Phoenix and p1 in Cambridge.
note     It will be discarded by Phoenix because Phoenix has no record that
note     p1 in Cambridge is waiting for dbo1 in Phoenix since control
note     message 1 still has not been received.
rcvcm 3
  Control message number  3  representing an OBPL  has been received.
rcvcm 1
  Control message number  1  representing a remote  resource request
           has been received
  Resource not available, process remains blocked.
  Control message number  4  sent from  Phoenix       to  Cambridge
           representing an OBPL
note     This OBPL contains entries for p1 in Cambridge and p1 in Phoenix.
note     It states that p1 in Phoenix is waiting for a message in message
note     group mg2.  Cambridge will verify that the desired message has
note     not been sent, and a deadlock will be detected.
rcvcm 4
  Control message number  4  representing an OBPL  has been received.
  A deadlock has been detected.  The following processes are involved:
                      p1            at node   Cambridge
                      p1            at node   Phoenix
           End of deadlock list
```



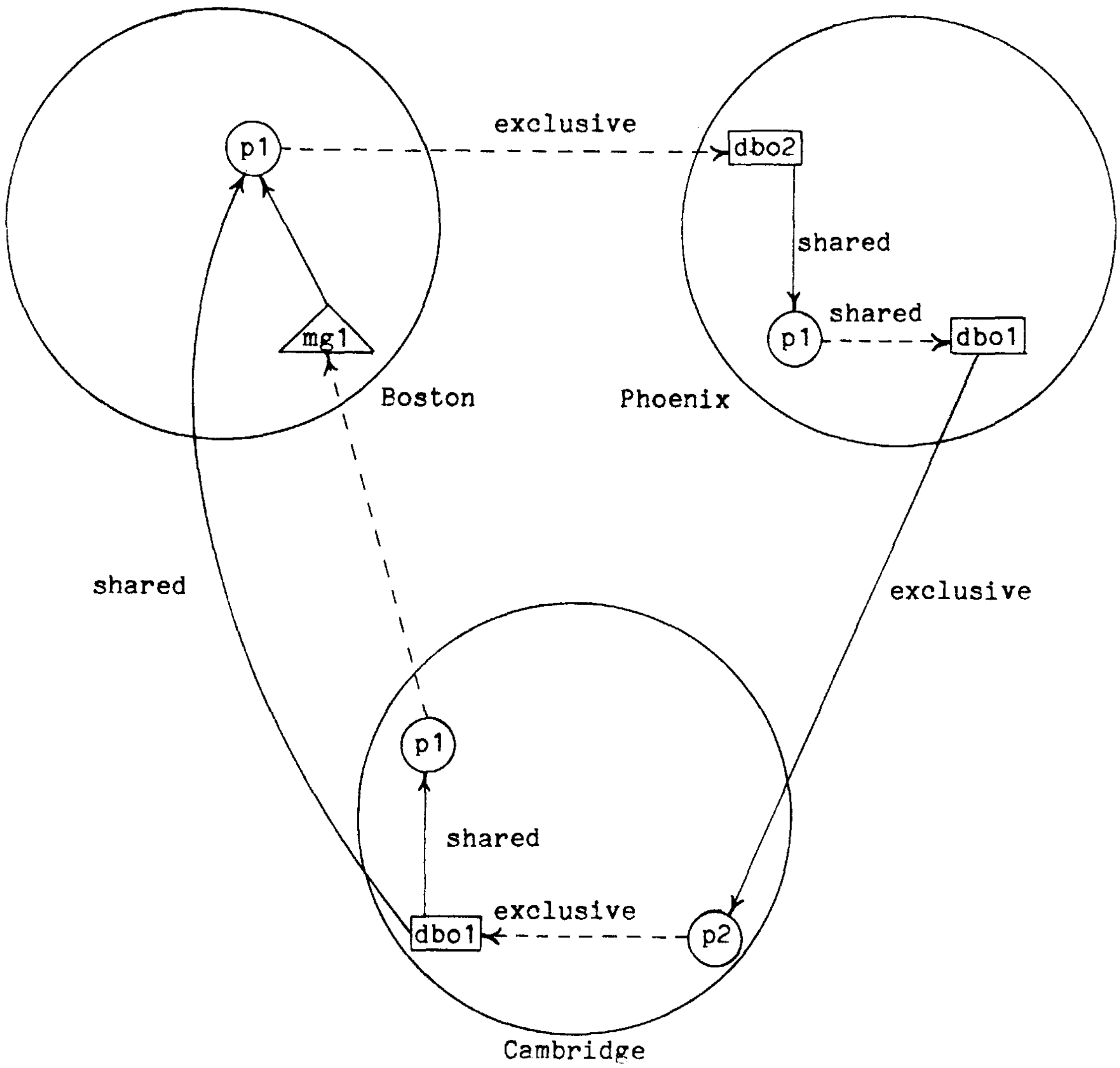
Final State Diagram

```
scenario demo3
note     This is an example of a two process two resource deadlock
note     involving two nodes.  The first three commands create the state
note     where both processes are active and both involved resources have
note     been allocated to the proper processes.
rqdbo exclusive Phoenix p1 Phoenix dbo1
  p1  at node  Phoenix  is granted exclusive use of  dbo1  at node  Phoenix
initmg mg2 Cambridge p1 Phoenix
  Message group   mg2  has been initiated
acceptmg mg2 Phoenix p1
  mg2   has been accepted by   p1   at node   Phoenix
rqdbo shared Cambridge p1 Phoenix dbo1
  Process  p1  at node  Cambridge  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  1  sent from  Cambridge  to  Phoenix
           representing a remote resource request
note     We will delay receipt by Phoenix of this resource request just
note     long enough to block p1 in Phoenix (which controls dbo1 in Phoenix)
note     and send an OBPL to Cambridge.  In this way, after receipt of the
note     resource request, we will have two OBPL's outstanding, and the same
note     deadlock will be detected twice.
rcvmsg mg2
  Process   p1             at node  Phoenix       is blocked waiting for a
           message in message group  mg2
  Control message number  2  sent from  Phoenix       to  Cambridge
           representing an OBPL
rcvcm 1
  Control message number  1  representing a remote  resource request
           has been received
  Resource not available, process remains blocked.
  Control message number  3  sent from  Phoenix       to  Cambridge
           representing an OBPL
rcvcm 2
  Control message number  2  representing an OBPL  has been received.
  Control message number  4  sent from  Cambridge     to  Phoenix
           representing an OBPL
rcvcm 3
  Control message number  3  representing an OBPL  has been received.
  A deadlock has been detected.  The following processes are involved:
                    p1             at node   Cambridge
                    p1             at node   Phoenix
           End of deadlock list
rcvcm 4
  Control message number  4  representing an OBPL  has been received.
  A deadlock has been detected.  The following processes are involved:
                    p1             at node   Phoenix
                    p1             at node   Cambridge
           End of deadlock list
```

```
scenario demo4
note     This is an example of a two process two resource deadlock
note     involving two nodes.  The first three commands create the state
note     where both processes are active and both involved resources have
note     been allocated to the proper processes.
rqdbo exclusive Phoenix p1 Phoenix dbo1
  p1  at node  Phoenix  is granted exclusive use of  dbo1  at node  Phoenix
initmg mg2 Cambridge p1 Phoenix
  Message group  mg2  has been initiated
acceptmg mg2 Phoenix p1
  mg2  has been accepted by   p1   at node   Phoenix
rqdbo shared Cambridge p1 Phoenix dbo1
  Process  p1  at node  Cambridge  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  1  sent from  Cambridge  to  Phoenix
           representing a remote resource request
note     We will allow this resource request to be immediately received
note     by Phoenix.  No OBPL will be generated because p1 in Phoenix is
note     active, and it controls dbo1 in Phoenix.  By default, control
note     messages generated in the future will be received immediately
note     after they are sent, and the deadlock will be detected once.
rcvcm 1
  Control message number  1  representing a remote  resource request
           has been received
  Resource not available, process remains blocked.
rcvmsg mg2
  Process   p1           at node  Phoenix       is blocked waiting for a
           message in message group  mg2
  Control message number  2  sent from  Phoenix       to  Cambridge
           representing an OBPL
rcvcm 2
  Control message number  2  representing an OBPL  has been received.
  Control message number  3  sent from  Cambridge     to  Phoenix
           representing an OBPL
rcvcm 3
  Control message number  3  representing an OBPL  has been received.
  A deadlock has been detected.  The following processes are involved:
                    p1             at node   Phoenix
                    p1             at node   Cambridge
           End of deadlock list
```

```
scenario demo5
note      This is an example of a state where two deadlocks exist
note      involving four processes and four resources located in three
note      nodes.  Two deadlocks are involved because dbo1 in Cambridge
note      has two shared users.  The first 10 commands create the state
note      where all the involved processes are active and all the involved
note      resources have been allocated to the proper processes.
initmg mg1 Boston p1 Cambridge
  Message group   mg1  has been initiated
acceptmg mg1 Cambridge p1
  mg1   has been accepted by   p1   at node   Cambridge
rqdbo shared Cambridge p1 Cambridge dbo1
  p1  at node  Cambridge  granted shared access to  dbo1  at node  Cambridge
rqdbo shared Boston p1 Cambridge dbo1
  Process  p1  at node  Boston  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  1  sent from  Boston  to  Cambridge
           representing a remote resource request
rcvcm 1
  Control message number  1  representing a remote  resource request
           has been received
  p1          at node  Boston        is granted shared access to
           dbo1          at node  Cambridge
  Control message number  2  sent from  Cambridge     to  Boston
           representing this allocation
rcvcm 2
  Control message number  2  representing a remote  resource allocation
           has been received
  p1          at node  Boston        has been granted shared access to
           dbo1          at node  Cambridge
rqdbo exclusive Cambridge p2 Phoenix dbo1
  Process  p2  at node  Cambridge  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  3  sent from  Cambridge  to  Phoenix
           representing a remote resource request
rcvcm 3
  Control message number  3  representing a remote  resource request
           has been received
  p2          at node  Cambridge     is granted exclusive use of
           dbo1          at node  Phoenix
  Control message number  4  sent from  Phoenix       to  Cambridge
           representing this allocation
rcvcm 4
  Control message number  4  representing a remote  resource allocation
           has been received
  p2          at node  Cambridge     has been granted exclusive use of
           dbo1          at node  Phoenix
rqdbo shared Phoenix p1 Phoenix dbo2
  p1  at node  Phoenix  granted shared access to  dbo2  at node  Phoenix
rqdbo exclusive Boston p1 Phoenix dbo2
  Process  p1  at node  Boston  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  5  sent from  Boston  to  Phoenix
           representing a remote resource request
note      No OBPL will be sent to another node, and no deadlock will
note      be detected because p1 at node Phoenix is active and is the only
note      process that has access to dbo2 in Phoenix.
rcvcm 5
  Control message number  5  representing a remote  resource request
           has been received
  Resource is not currently available for  exclusive use, process  p1
           at node  Boston        remains blocked
rqdbo shared Phoenix p1 Phoenix dbo1
  Resource not available, process blocked.
  Control message number  6  sent from  Phoenix       to  Cambridge
           representing an OBPL
```

```
note     No deadlock will be detected because p2 in Cambridge is active.
rcvcm 6
  Control message number  6  representing an OBPL  has been received.
note     This next request will create a three process three resource
note     deadlock.  An OBPL will be created, and we will immediately pass
note     it from node to node in order to detect the deadlock.
rqdbo exclusive Cambridge p2 Cambridge dbo1
  Resource is not currently available for  exclusive use, process  p2
          at node  Cambridge  is blocked.
  Control message number  7  sent from  Cambridge     to  Boston
           representing an OBPL
rcvcm 7
  Control message number  7  representing an OBPL  has been received.
  Control message number  8  sent from  Boston        to  Phoenix
           representing an OBPL
rcvcm 8
  Control message number  8  representing an OBPL  has been received.
  A deadlock has been detected.  The following processes are involved:
                    p2            at node   Cambridge
                    p1            at node   Boston
                    p1            at node   Phoenix
           End of deadlock list
note     The next command will create a four process four resource deadlock.
note     Due to the fact that two processes have shared access to dbo1 in
note     Cambridge, both this newly created deadlock, and the previously
note     detected deadlock will be detected when the OBPL is created and
note     passed among the nodes.
rcvmsg mg1
  Process   p1           at node  Cambridge     is blocked waiting for a
           message in message group  mg1
  Control message number  9  sent from  Cambridge     to  Boston
           representing an OBPL
rcvcm 9
  Control message number  9  representing an OBPL  has been received.
  Control message number  10  sent from  Boston        to  Phoenix
           representing an OBPL
rcvcm 10
  Control message number  10  representing an OBPL  has been received.
  Control message number  11  sent from  Phoenix       to  Cambridge
           representing an OBPL
rcvcm 11
  Control message number  11  representing an OBPL  has been received.
  A deadlock has been detected.  The following processes are involved:
                    p1            at node   Cambridge
                    p1            at node   Boston
                    p1            at node   Phoenix
                    p2            at node   Cambridge
           End of deadlock list
  A deadlock has been detected.  The following processes are involved:
                    p1            at node   Boston
                    p1            at node   Phoenix
                    p2            at node   Cambridge
           End of deadlock list
```



Final State Diagram

```
scenario demo6
note     This is an example of a state where two deadlocks exist
note     involving four processes and four resources located in three
note     nodes.  Two deadlocks are involved because dbo1 in Cambridge
note     has two shared users.  The first 10 commands create the state
note     where all the involved processes are active and all the involved
note     resources have been allocated to the proper processes.
initmg mg1 Boston p1 Cambridge
  Message group   mg1  has been initiated
acceptmg mg1 Cambridge p1
  mg1   has been accepted by   p1   at node   Cambridge
rqdbo shared Cambridge p1 Cambridge dbo1
  p1  at node  Cambridge  granted shared access to  dbo1  at node  Cambridge
rqdbo shared Boston p1 Cambridge dbo1
  Process  p1  at node  Boston  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  1  sent from  Boston  to  Cambridge
           representing a remote resource request
rcvcm 1
  Control message number  1  representing a remote  resource request
           has been received
  p1            at node  Boston         is granted shared access to
          dbo1           at node  Cambridge
  Control message number  2  sent from  Cambridge    to  Boston
           representing this allocation
rcvcm 2
  Control message number  2  representing a remote  resource allocation
           has been received
  p1            at node  Boston         has been granted shared access to
           dbo1          at node  Cambridge
rqdbo exclusive Cambridge p2 Phoenix dbo1
  Process  p2  at node  Cambridge  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  3  sent from  Cambridge  to  Phoenix
           representing a remote resource request
rcvcm 3
  Control message number  3  representing a remote  resource request
           has been received
  p2            at node  Cambridge      is granted exclusive use of
           dbo1          at node  Phoenix
  Control message number  4  sent from  Phoenix      to  Cambridge
           representing this allocation
rcvcm 4
  Control message number  4  representing a remote  resource allocation
           has been received
  p2            at node  Cambridge      has been granted exclusive use of
           dbo1          at node  Phoenix
rqdbo shared Phoenix p1 Phoenix dbo2
  p1  at node  Phoenix  granted shared access to  dbo2  at node  Phoenix
rqdbo exclusive Boston p1 Phoenix dbo2
  Process  p1  at node  Boston  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  5  sent from  Boston  to  Phoenix
           representing a remote resource request
note     p1 in Phoenix is active, so there will be no deadlock when the
note     remote resource request is received from Boston.
rcvcm 5
  Control message number  5  representing a remote  resource request
           has been received
  Resource is not currently available for  exclusive use, process  p1
           at node  Boston          remains blocked
rqdbo shared Phoenix p1 Phoenix dbo1
  Resource not available, process blocked.
  Control message number  6  sent from  Phoenix      to  Cambridge
           representing an OBPL
```

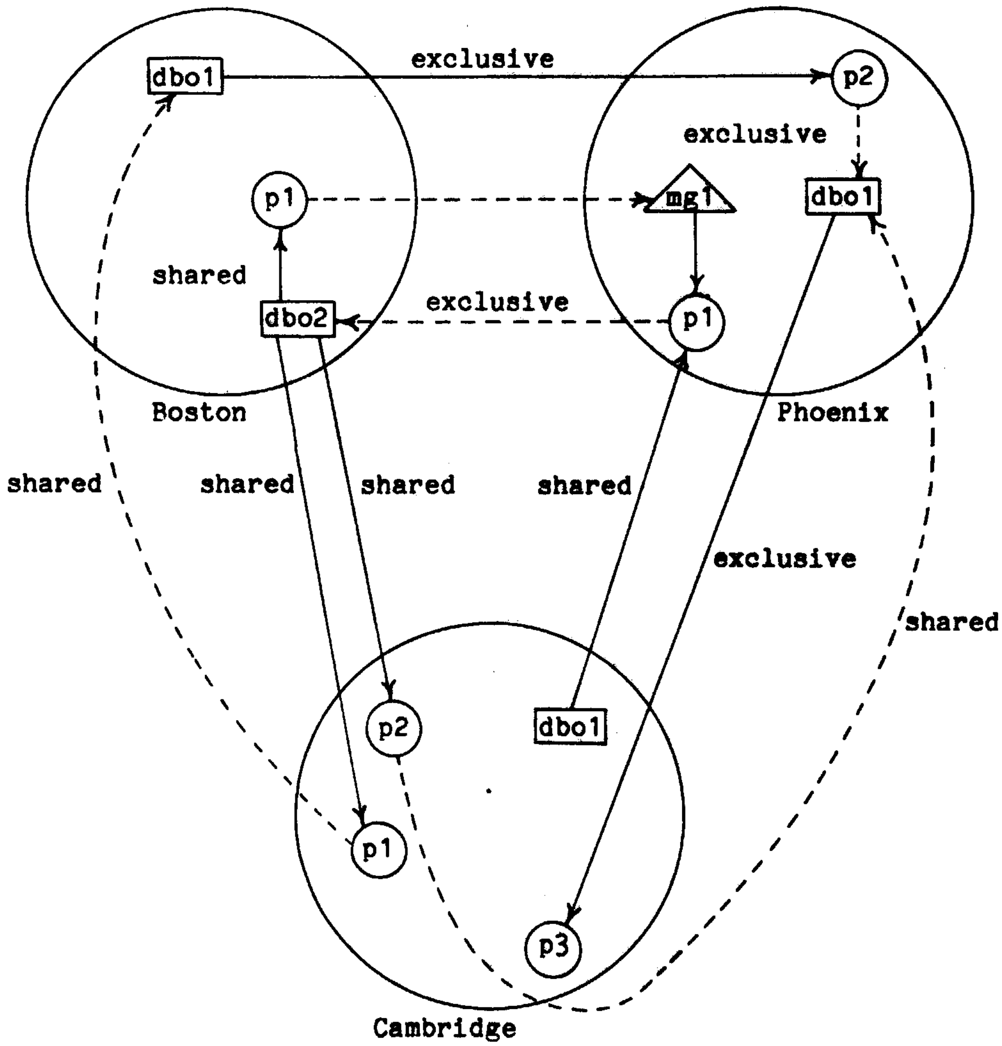
```
note     p2 in Cambridge is active, so the OBPL will be discarded after
note     it is received by Cambridge.
rcvcm 6
  Control message number  6  representing an OBPL  has been received.
rcvmsg mg1
  Process   p1             at node  Cambridge     is blocked waiting for a
           message in message group  mg1
  Control message number  7  sent from  Cambridge     to  Boston
           representing an OBPL
note     p2 in Cambridge is active, so the OBPL will be discarded when
note     it reaches Cambridge.
rcvcm 7
  Control message number  7  representing an OBPL  has been received.
  Control message number  8  sent from  Boston        to  Phoenix
           representing an OBPL
rcvcm 8
  Control message number  8  representing an OBPL  has been received.
  Control message number  9  sent from  Phoenix       to  Cambridge
           representing an OBPL
rcvcm 9
  Control message number  9  representing an OBPL  has been received.
note     This next request will create two deadlocks, due to the fact that
note     dbo1 in Cambridge has two readers.  Two OBPL's will be generated,
note     and both deadlocks will be detected when their respective OBPL's
note     arrive in Phoenix.  The OBPL's need not return to Cambridge
note     because p2 in Cambridge was the first process to be placed in the
note     OBPL's, and Phoenix knows that p2 in Cambridge controls dbo1
note     in Phoenix.
rqdbo exclusive Cambridge p2 Cambridge dbo1
  Resource is not currently available for  exclusive use, process  p2
           at node  Cambridge  is blocked.
  Control message number  10  sent from  Cambridge     to  Boston
           representing an OBPL
  Control message number  11  sent from  Cambridge     to  Boston
           representing an OBPL
rcvcm 10
  Control message number  10  representing an OBPL  has been received.
  Control message number  12  sent from  Boston        to  Phoenix
           representing an OBPL
rcvcm 12
  Control message number  12  representing an OBPL  has been received.
  A deadlock has been detected.  The following processes are involved:
                     p2             at node   Cambridge
                     p1             at node   Cambridge
                     p1             at node   Boston
                     p1             at node   Phoenix
           End of deadlock list
rcvcm 11
  Control message number  11  representing an OBPL  has been received.
  Control message number  13  sent from  Boston        to  Phoenix
           representing an OBPL
rcvcm 13
  Control message number  13  representing an OBPL  has been received.
  A deadlock has been detected.  The following processes are involved:
                     p2             at node   Cambridge
                     p1             at node   Boston
                     p1             at node   Phoenix
           End of deadlock list
```

```
scenario demo7
note     This is an example of a state where three deadlocks exist
note     involving six processes and five resources located in three
note     nodes. Three deadlocks are involved because dbo2 in Boston
note     has three shared users. Five, rather than six, resources are
note     involved because two processes are waiting for the same database
note     object. The first 18 commands create the state where all the
note     involved processes are active and all the involved resources
note     have been allocated to the proper processes.
rqdbo shared Boston p1 Boston dbo2
  p1  at node  Boston  granted shared access to  dbo2  at node  Boston
initmg mg1 Phoenix p1 Boston
  Message group  mg1  has been initiated
acceptmg mg1 Boston p1
  mg1  has been accepted by  p1  at node  Boston
rqdbo exclusive Phoenix p2 Boston dbo1
  Process  p2  at node  Phoenix  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  1  sent from  Phoenix  to  Boston
           representing a remote resource request
rcvcm 1
  Control message number  1  representing a remote  resource request
           has been received
  p2          at node  Phoenix       is granted exclusive use of
           dbo1          at node  Boston
  Control message number  2  sent from  Boston        to  Phoenix
           representing this allocation
rcvcm 2
  Control message number  2  representing a remote  resource allocation
           has been received
  p2          at node  Phoenix       has been granted exclusive use of
           dbo1          at node  Boston
rqdbo shared Cambridge p1 Boston dbo2
  Process  p1  at node  Cambridge  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  3  sent from  Cambridge  to  Boston
           representing a remote resource request
rcvcm 3
  Control message number  3  representing a remote  resource request
           has been received
  p1          at node  Cambridge     is granted shared access to
           dbo2          at node  Boston
  Control message number  4  sent from  Boston        to  Cambridge
           representing this allocation
rcvcm 4
  Control message number  4  representing a remote  resource allocation
           has been received
  p1          at node  Cambridge     has been granted shared access to
           dbo2          at node  Boston
rqdbo shared Cambridge p2 Boston dbo2
  Process  p2  at node  Cambridge  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  5  sent from  Cambridge  to  Boston
           representing a remote resource request
rqdbo shared Phoenix p1 Cambridge dbo1
  Process  p1  at node  Phoenix  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  6  sent from  Phoenix  to  Cambridge
           representing a remote resource request
rcvcm 5
  Control message number  5  representing a remote  resource request
           has been received
  p2          at node  Cambridge     is granted shared access to
           dbo2          at node  Boston
  Control message number  7  sent from  Boston        to  Cambridge
           representing this allocation
```

```
rcvcm 6
  Control message number  6  representing a remote  resource request
          has been received
  p1          at node  Phoenix       is granted shared access to
          dbo1          at node  Cambridge
  Control message number  8  sent from  Cambridge     to  Phoenix
          representing this allocation
rcvcm 7
  Control message number  7  representing a remote  resource allocation
          has been received
  p2          at node  Cambridge     has been granted shared access to
          dbo2          at node  Boston
rcvcm 8
  Control message number  8  representing a remote  resource allocation
          has been received
  p1          at node  Phoenix       has been granted shared access to
          dbo1          at node  Cambridge
rqdbo exclusive Cambridge p3 Phoenix dbo1
  Process  p3  at node  Cambridge  is blocked while a request is sent to
          the node containing the desired resource
  Control message number  9  sent from  Cambridge  to  Phoenix
          representing a remote resource request
rcvcm 9
  Control message number  9  representing a remote  resource request
          has been received
  p3          at node  Cambridge     is granted exclusive use of
          dbo1          at node  Phoenix
  Control message number  10  sent from  Phoenix       to  Cambridge
          representing this allocation
rcvcm 10
  Control message number  10  representing a remote  resource allocation
          has been received
  p3          at node  Cambridge     has been granted exclusive use of
          dbo1          at node  Phoenix
rcvmsg mg1
  Process   p1          at node  Boston        is blocked waiting for a
          message in message group  mg1
  Control message number  11  sent from  Boston        to  Phoenix
          representing an OBPL
note      The OBPL will be discarded by Phoenix because p1 is active.
rcvcm 11
  Control message number  11  representing an OBPL  has been received.
rqdbo shared Cambridge p1 Boston dbo1
  Process  p1  at node  Cambridge  is blocked while a request is sent to
          the node containing the desired resource
  Control message number  12  sent from  Cambridge  to  Boston
          representing a remote resource request
note      The process that controls dbo1 in Boston is located in Phoenix,
note      and is active.  Therefore, when Boston receives the resource
note      request, it will create an OBPL and send it to Phoenix, which
note      will then discard it.
rcvcm 12
  Control message number  12  representing a remote  resource request
          has been received
  Resource not available, process remains blocked.
  Control message number  13  sent from  Boston        to  Phoenix
          representing an OBPL
rcvcm 13
  Control message number  13  representing an OBPL  has been received.
rqdbo exclusive Phoenix p2 Phoenix dbo1
  Resource not available, process blocked.
  Control message number  14  sent from  Phoenix       to  Cambridge
          representing an OBPL
note      The OBPL will be discarded by Cambridge because p3, which controls
note      dbo1 in Phoenix, is active.
rcvcm 14
  Control message number  14  representing an OBPL  has been received.
```

```
rqdbo exclusive Cambridge p3 Cambridge dbo1
  Resource is not currently available for  exclusive use, process  p3
           at node  Cambridge  is blocked.
  Control message number  15  sent from  Cambridge     to  Phoenix
           representing an OBPL
note     The OBPL will be discarded by Phoenix because p1, which controls
note     dbo1 in Cambridge, is active.
rcvcm 15
  Control message number  15  representing an OBPL  has been received.
rqdbo shared Cambridge p2 Phoenix dbo1
  Process  p2  at node  Cambridge  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  16  sent from  Cambridge  to  Phoenix
           representing a remote resource request
rcvcm 16
  Control message number  16  representing a remote  resource request
           has been received
  Resource not available, process remains blocked.
  Control message number  17  sent from  Phoenix       to  Cambridge
           representing an OBPL
note     An OBPL is sent to Cambridge because p3 in Cambridge controls
note     dbo1 in Phoenix.  p3 will be added to the OBPL which will then
note     be passed to Phoenix because p1 in Phoenix controls dbo1 in
note     Cambridge.  The OBPL will then be discarded because p1 is active.
rcvcm 17
  Control message number  17  representing an OBPL  has been received.
  Control message number  18  sent from  Cambridge     to  Phoenix
           representing an OBPL
rcvcm 18
  Control message number  18  representing an OBPL  has been received.
note     The next request creates three deadlocks.  When Boston receives
note     the remote resource request for dbo2, it creates three OBPL's
note     because there are three readers of the database object.  We will
note     then allow the three OBPL's to be passed among nodes until all
note     three deadlocks have been detected, at which time there will be
note     no outstanding OBPL's or control messages.
rqdbo exclusive Phoenix p1 Boston dbo2
  Process  p1  at node  Phoenix  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  19  sent from  Phoenix  to  Boston
           representing a remote resource request
rcvcm 19
  Control message number  19  representing a remote  resource request
           has been received
  Resource is not currently available for  exclusive use, process  p1
           at node  Phoenix       remains blocked
  Control message number  20  sent from  Boston        to  Cambridge
           representing an OBPL
  Control message number  21  sent from  Boston        to  Phoenix
           representing an OBPL
  Control message number  22  sent from  Boston        to  Cambridge
           representing an OBPL
rcvcm 21
  Control message number  21  representing an OBPL  has been received.
  A deadlock has been detected.  The following processes are involved:
                     p1            at node   Phoenix
                     p1            at node   Boston
           End of deadlock list
rcvcm 20
  Control message number  20  representing an OBPL  has been received.
  Control message number  23  sent from  Cambridge     to  Boston
           representing an OBPL
rcvcm 22
  Control message number  22  representing an OBPL  has been received.
  Control message number  24  sent from  Cambridge     to  Phoenix
           representing an OBPL
```

```
rcvcm 23
  Control message number  23  representing an OBPL  has been received.
  Control message number  25  sent from  Boston        to  Phoenix
          representing an OBPL
rcvcm 25
  Control message number  25  representing an OBPL  has been received.
  Control message number  26  sent from  Phoenix       to  Cambridge
          representing an OBPL
rcvcm 26
  Control message number  26  representing an OBPL  has been received.
  A deadlock has been detected.  The following processes are involved:
                    p1            at node   Phoenix
                    p1            at node   Cambridge
                    p2            at node   Phoenix
                    p3            at node   Cambridge
          End of deadlock list
rcvcm 24
  Control message number  24  representing an OBPL  has been received.
  Control message number  27  sent from  Phoenix       to  Cambridge
          representing an OBPL
rcvcm 27
  Control message number  27  representing an OBPL  has been received.
  A deadlock has been detected.  The following processes are involved:
                    p1            at node   Phoenix
                    p2            at node   Cambridge
                    p3            at node   Cambridge
          End of deadlock list
```



Final State Diagram

```
scenario demo8
note     This is an example of a state where three deadlocks exist
note     involving six processes and five resources located in three
note     nodes. Three deadlocks are involved because dbo2 in Boston
note     has three shared users. Five, rather than six, resources are
note     involved because two processes are waiting for the same database
note     object. The first 18 commands create the state where all the
note     involved processes are active and all the involved resources
note     have been allocated to the proper processes.
rqdbo shared Boston p1 Boston dbo2
  p1  at node  Boston  granted shared access to  dbo2  at node  Boston
initmg mg1 Phoenix p1 Boston
  Message group  mg1  has been initiated
acceptmg mg1 Boston p1
  mg1  has been accepted by  p1  at node  Boston
rqdbo exclusive Phoenix p2 Boston dbo1
  Process  p2  at node  Phoenix  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  1  sent from  Phoenix  to  Boston
           representing a remote resource request
rcvcm 1
  Control message number  1  representing a remote  resource request
           has been received
  p2          at node  Phoenix       is granted exclusive use of
           dbo1         at node  Boston
  Control message number  2  sent from  Boston         to  Phoenix
           representing this allocation
rcvcm 2
  Control message number  2  representing a remote  resource allocation
           has been received
  p2          at node  Phoenix       has been granted exclusive use of
           dbo1         at node  Boston
rqdbo shared Cambridge p1 Boston dbo2
  Process  p1  at node  Cambridge  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  3  sent from  Cambridge  to  Boston
           representing a remote resource request
rcvcm 3
  Control message number  3  representing a remote  resource request
           has been received
  p1          at node  Cambridge     is granted shared access to
           dbo2         at node  Boston
  Control message number  4  sent from  Boston         to  Cambridge
           representing this allocation
rcvcm 4
  Control message number  4  representing a remote  resource allocation
           has been received
  p1          at node  Cambridge     has been granted shared access to
           dbo2         at node  Boston
rqdbo shared Cambridge p2 Boston dbo2
  Process  p2  at node  Cambridge  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  5  sent from  Cambridge  to  Boston
           representing a remote resource request
rqdbo shared Phoenix p1 Cambridge dbo1
  Process  p1  at node  Phoenix  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  6  sent from  Phoenix  to  Cambridge
           representing a remote resource request
rcvcm 5
  Control message number  5  representing a remote  resource request
           has been received
  p2          at node  Cambridge     is granted shared access to
           dbo2         at node  Boston
  Control message number  7  sent from  Boston         to  Cambridge
           representing this allocation
```

```
rcvcm 6
  Control message number  6  representing a remote  resource request
          has been received
  p1          at node  Phoenix       is granted shared access to
          dbo1          at node  Cambridge
  Control message number  8  sent from  Cambridge     to  Phoenix
          representing this allocation
rcvcm 7
  Control message number  7  representing a remote  resource allocation
          has been received
  p2          at node  Cambridge     has been granted shared access to
          dbo2          at node  Boston
rcvcm 8
  Control message number  8  representing a remote  resource allocation
          has been received
  p1          at node  Phoenix       has been granted shared access to
          dbo1          at node  Cambridge
rqdbo exclusive Cambridge p3 Phoenix dbo1
  Process  p3  at node  Cambridge  is blocked while a request is sent to
          the node containing the desired resource
  Control message number  9  sent from  Cambridge  to  Phoenix
          representing a remote resource request
rcvcm 9
  Control message number  9  representing a remote  resource request
          has been received
  p3          at node  Cambridge     is granted exclusive use of
          dbo1          at node  Phoenix
  Control message number  10  sent from  Phoenix       to  Cambridge
          representing this allocation
rcvcm 10
  Control message number  10  representing a remote  resource allocation
          has been received
  p3          at node  Cambridge     has been granted exclusive use of
          dbo1          at node  Phoenix
rqdbo exclusive Phoenix p1 Boston dbo2
  Process  p1  at node  Phoenix  is blocked while a request is sent to
          the node containing the desired resource
  Control message number  11  sent from  Phoenix  to  Boston
          representing a remote resource request
note     After receipt of the remote resource request, Boston will send
note     two OBPL's to Cambridge because two processes in that node have
note     shared use of dbo2 in Boston.  A third external message is not
note     needed because the third reader of dbo2 is located in Boston
note     and is active.  We will delay the receipt of one of the OBPL's
note     until after the process in the list that controls dbo2 gets
note     blocked waiting for a resource located in Phoenix.
rcvcm 11
  Control message number  11  representing a remote  resource request
          has been received
  Resource is not currently available for  exclusive use, process  p1
          at node  Phoenix        remains blocked
  Control message number  12  sent from  Boston        to  Cambridge
          representing an OBPL
  Control message number  13  sent from  Boston        to  Cambridge
          representing an OBPL
rcvcm 12
  Control message number  12  representing an OBPL  has been received.
rqdbo shared Cambridge p1 Boston dbo1
  Process  p1  at node  Cambridge  is blocked while a request is sent to
          the node containing the desired resource
  Control message number  14  sent from  Cambridge  to  Boston
          representing a remote resource request
rqdbo shared Cambridge p2 Phoenix dbo1
  Process  p2  at node  Cambridge  is blocked while a request is sent to
          the node containing the desired resource
  Control message number  15  sent from  Cambridge  to  Phoenix
          representing a remote resource request
```

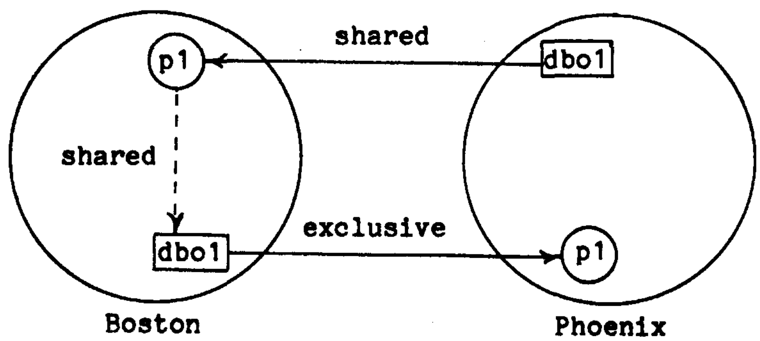
```
rcvcm 13
  Control message number  13  representing an OBPL  has been received.
  Control message number  16  sent from  Cambridge      to  Phoenix
          representing an OBPL
note     Let Phoenix receive the OBPL before it receives the remote resource
note     request that was assumed to have taken place before the last
note     process was added to the OBPL.  The OBPL will be discarded because
note     Phoenix has no record that p2 in Cambridge is waiting for dbo1
note     in Phoenix.
rcvcm 16
  Control message number  16  representing an OBPL  has been received.
note     Now let the above mentioned remote resource request be received
note     by Phoenix.  An OBPL will be created and sent to Cambridge, which
note     will then discard the OBPL because p3 is active.
rcvcm 15
  Control message number  15  representing a remote  resource request
          has been received
  Resource not available, process remains blocked.
  Control message number  17  sent from  Phoenix        to  Cambridge
          representing an OBPL
rcvcm 17
  Control message number  17  representing an OBPL  has been received.
note     Now let the remote resource request for dbo1 in Boston by p1 in
note     Cambridge be received by Boston.  An OBPL will be created and sent
note     to Phoenix, where p2 in Phoenix is waiting for dbo1 in Phoenix, so
note     the OBPL will be passed on to Cambridge where p3 is active, and
note     the OBPL will then be discarded.
rcvcm 14
  Control message number  14  representing a remote  resource request
          has been received
  Resource not available, process remains blocked.
  Control message number  18  sent from  Boston         to  Phoenix
          representing an OBPL
rcvcm 18
  Control message number  18  representing an OBPL  has been received.
rqdbo exclusive Phoenix p2 Phoenix dbo1
  Resource not available, process blocked.
  Control message number  19  sent from  Phoenix        to  Cambridge
          representing an OBPL
note     The OBPL will be discarded by Cambridge because p3 is active.
rcvcm 19
  Control message number  19  representing an OBPL  has been received.
note     The next command will create a two process two resource deadlock.
note     An OBPL will be sent to Phoenix, which will append p1 in Phoenix
note     to the OBPL and send the OBPL back to Boston because p1 is waiting
note     for dbo2 in Boston.  The deadlock will then be detected, and two
note     OBPL's will be sent to Cambridge because there are three readers
note     of dbo2.  These OBPL's will then be passed around until they
note     return to Cambridge, where they will be discarded because p3 in
note     Cambridge will still be active when the OBPL's get examined.
rcvmsg mg1
  Process    p1          at node  Boston         is blocked waiting for a
          message in message group  mg1
  Control message number  20  sent from  Boston         to  Phoenix
          representing an OBPL
rcvcm 20
  Control message number  20  representing an OBPL  has been received.
  Control message number  21  sent from  Phoenix        to  Boston
          representing an OBPL
```

```
rcvcm 21
  Control message number  21  representing an OBPL  has been received.
  Control message number  22  sent from  Boston        to  Cambridge
          representing an OBPL
  A deadlock has been detected.  The following processes are involved:
                   p1            at node   Boston
                   p1            at node   Phoenix
          End of deadlock list
  Control message number  23  sent from  Boston        to  Cambridge
          representing an OBPL
rcvcm 22
  Control message number  22  representing an OBPL  has been received.
  Control message number  24  sent from  Cambridge     to  Boston
          representing an OBPL
rcvcm 24
  Control message number  24  representing an OBPL  has been received.
  Control message number  25  sent from  Boston        to  Phoenix
          representing an OBPL
rcvcm 25
  Control message number  25  representing an OBPL  has been received.
  Control message number  26  sent from  Phoenix       to  Cambridge
          representing an OBPL
rcvcm 26
  Control message number  26  representing an OBPL  has been received.
rcvcm 23
  Control message number  23  representing an OBPL  has been received.
  Control message number  27  sent from  Cambridge     to  Phoenix
          representing an OBPL
rcvcm 27
  Control message number  27  representing an OBPL  has been received.
  Control message number  28  sent from  Phoenix       to  Cambridge
          representing an OBPL
rcvcm 28
  Control message number  28  representing an OBPL  has been received.
note     This next request will create two deadlocks.  An OBPL will be
note     sent to Phoenix, which will add p1 in Phoenix to the list and
note     send it to Boston.   Boston will then send out three OBPL'S,
note     one for each reader of dbo2 in Boston.  These OBPL's will be
note     passed among the various nodes until there are no more OBPL's
note     and control messages outstanding.  Note that the two process two
note     resource deadlock will be detected for a second time because of
note     the fact that p1 in Boston still has shared access to dbo2 in
note     Boston and the deadlock has not been broken by aborting any
note     processes.
rqdbo exclusive Cambridge p3 Cambridge dbo1
  Resource is not currently available for  exclusive use, process  p3
          at node  Cambridge  is blocked.
  Control message number  29  sent from  Cambridge     to  Phoenix
          representing an OBPL
rcvcm 29
  Control message number  29  representing an OBPL  has been received.
  Control message number  30  sent from  Phoenix       to  Boston
          representing an OBPL
rcvcm 30
  Control message number  30  representing an OBPL  has been received.
  Control message number  31  sent from  Boston        to  Cambridge
          representing an OBPL
  Control message number  32  sent from  Boston        to  Phoenix
          representing an OBPL
  Control message number  33  sent from  Boston        to  Cambridge
          representing an OBPL
rcvcm 32
  Control message number  32  representing an OBPL  has been received.
  A deadlock has been detected.  The following processes are involved:
                   p1            at node   Phoenix
                   p1            at node   Boston
          End of deadlock list
```

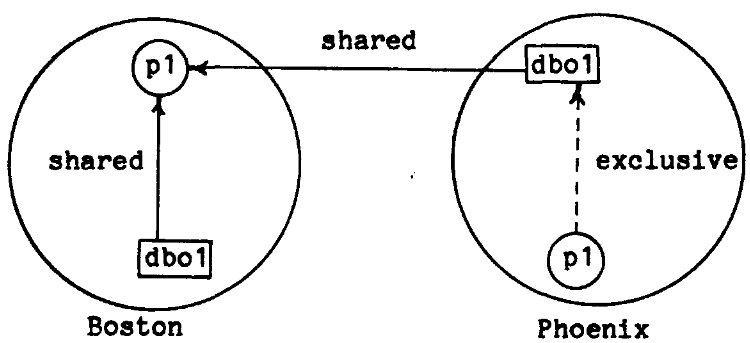
```
rcvcm 31
  Control message number  31  representing an OBPL  has been received.
  Control message number  34  sent from  Cambridge     to  Boston
          representing an OBPL
rcvcm 34
  Control message number  34  representing an OBPL  has been received.
  Control message number  35  sent from  Boston        to  Phoenix
          representing an OBPL
rcvcm 35
  Control message number  35  representing an OBPL  has been received.
  A deadlock has been detected.  The following processes are involved:
                    p3           at node   Cambridge
                    p1           at node   Phoenix
                    p1           at node   Cambridge
                    p2           at node   Phoenix
          End of deadlock list
rcvcm 33
  Control message number  33  representing an OBPL  has been received.
  Control message number  36  sent from  Cambridge     to  Phoenix
          representing an OBPL
rcvcm 36
  Control message number  36  representing an OBPL  has been received.
  A deadlock has been detected.  The following processes are involved:
                    p3           at node   Cambridge
                    p1           at node   Phoenix
                    p2           at node   Cambridge
          End of deadlock list
```

```
scenario demo9
note     This is an example of a case where a process releases a remote
note     database object and sends a remote resource control message at
note     the same time that an OBPL is sent to this node stating that some
note     other process is waiting for the resource mentioned above, which
note     is controlled by the first process mentioned above.  Before the
note     OBPL arrives, the first process gets blocked waiting for a resource
note     that is controlled by the process that was placed in the OBPL.
note     No deadlock is detected because the resource in question is no
note     longer controlled by the last process to be added to the OBPL.
rqdbo shared Boston p1 Phoenix dbo1
  Process  p1  at node  Boston  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  1  sent from  Boston  to  Phoenix
           representing a remote resource request
rcvcm 1
  Control message number  1  representing a remote  resource request
           has been received
  p1            at node  Boston        is granted shared access to
           dbo1          at node  Phoenix
  Control message number  2  sent from  Phoenix      to  Boston
           representing this allocation
rcvcm 2
  Control message number  2  representing a remote  resource allocation
           has been received
  p1            at node  Boston        has been granted shared access to
           dbo1          at node  Phoenix
rqdbo exclusive Phoenix p1 Boston dbo1
  Process  p1  at node  Phoenix  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  3  sent from  Phoenix  to  Boston
           representing a remote resource request
rcvcm 3
  Control message number  3  representing a remote  resource request
           has been received
  p1            at node  Phoenix       is granted exclusive use of
           dbo1          at node  Boston
  Control message number  4  sent from  Boston       to  Phoenix
           representing this allocation
rcvcm 4
  Control message number  4  representing a remote  resource allocation
           has been received
  p1            at node  Phoenix       has been granted exclusive use of
           dbo1          at node  Boston
rqdbo shared Boston p1 Boston dbo1
  Resource not available, process blocked.
  Control message number  5  sent from  Boston       to  Phoenix
           representing an OBPL
note     Let dbo1 in Boston be released by p1 in Phoenix, and let p1 in
note     Phoenix then get blocked waiting for dbo1 in Phoenix before the
note     OBPL from Boston is received by Phoenix.
rldbo Phoenix p1 Boston dbo1
  Control message number  6  sent from  Phoenix  to  Boston
           representing a remote resource release
rcvcm 6
  Control message number  6  representing a remote  resource release
           has been received
  dbo1          at node  Boston        has been released by
           p1            at node  Phoenix
  Process  p1            at node  Boston        is granted shared access to
           dbo1          at node  Boston
rqdbo exclusive Phoenix p1 Phoenix dbo1
  Resource is not currently available for  exclusive use, process  p1
           at node  Phoenix  is blocked.
  Control message number  7  sent from  Phoenix      to  Boston
           representing an OBPL
```

```
rcvcm 7
  Control message number  7  representing an OBPL  has been received.
note     No deadlock will be detected because Phoenix observes that p1 in
note     Phoenix no longer has access to dbo1 in Boston, and discards
note     the OBPL.
rcvcm 5
  Control message number  5  representing an OBPL  has been received.
```

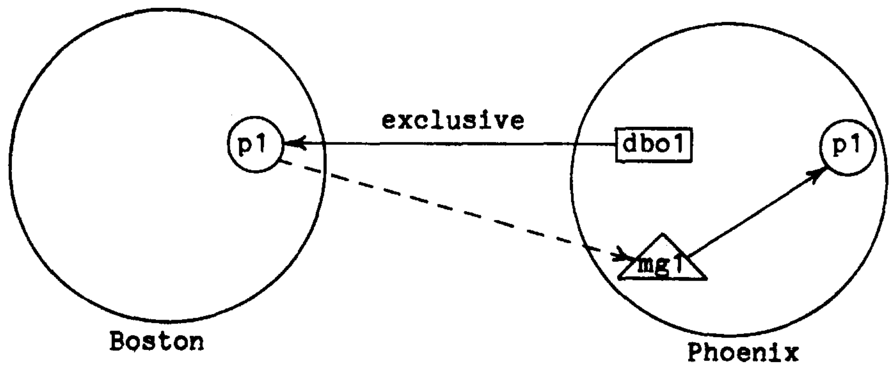


State where control message 5 has just been sent from Boston to Phoenix. Control message 5 represents an OBPL. Receipt of the OBPL is delayed until after the state drawn below is reached.

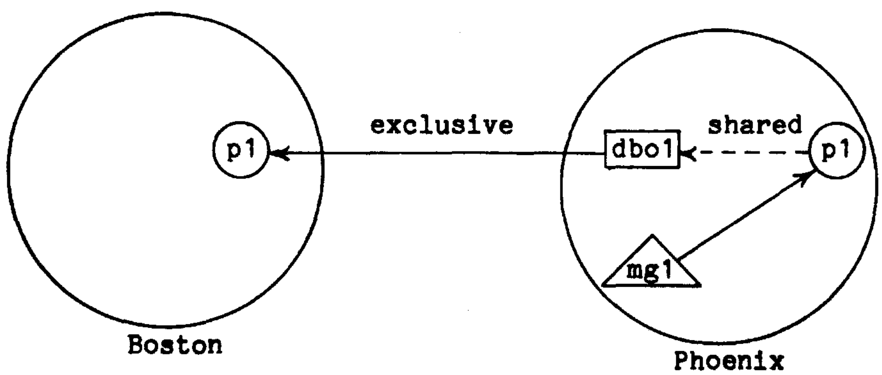


Final State Diagram

```
scenario demo10
note     This is an example where an OBPL is sent from Boston to Phoenix
note     stating that a process in Boston is waiting for a message from a
note     process in Phoenix.  Before the OBPL arrives in Phoenix, the
note     desired message is sent, and the process in Phoenix gets blocked
note     waiting for a resource that is controlled by the process that was
note     placed in the OBPL that was sent from Boston to Phoenix.  No
note     deadlock is detected because Phoenix notices that the message
note     that was desired by the process in Boston has already been sent.
note     The first six commands create the state where the OBPL
note     mentioned above has just been sent.
initmg mg1 Phoenix p1 Boston
  Message group   mg1  has been initiated
acceptmg mg1 Boston p1
  mg1  has been accepted by   p1   at node   Boston
rqdbo exclusive Boston p1 Phoenix dbo1
  Process  p1  at node  Boston  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  1  sent from  Boston  to  Phoenix
           representing a remote resource request
rcvcm 1
  Control message number  1  representing a remote  resource request
           has been received
  p1          at node  Boston         is granted exclusive use of
           dbo1          at node  Phoenix
  Control message number  2  sent from  Phoenix       to  Boston
           representing this allocation
rcvcm 2
  Control message number  2  representing a remote  resource allocation
           has been received
  p1          at node  Boston         has been granted exclusive use of
           dbo1          at node  Phoenix
rcvmsg mg1
  Process   p1             at node  Boston         is blocked waiting for a
           message in message group  mg1
  Control message number  3  sent from  Boston        to  Phoenix
           representing an OBPL
note     We will now temporarily delay receipt of the OBPL by Phoenix.
note     Send the message that the process in Boston desires.
sendmsg mg1
  Control message number   4   sent from   Phoenix        to   Boston
           representing a message in a message  group
note     Let the process in Boston receive the message.
rcvcm 4
  Control message_number  4  representing a message  in a message group
           has been received
  Process   p1             at node   Boston         has been awakened upon
           receipt of a message in message group  mg1
note     Block p1 in Phoenix and then let Boston discard the OBPL that
note     will be created as a result of this wait.
rqdbo shared Phoenix p1 Phoenix dbo1
  Resource not available, process blocked.
  Control message number  5  sent from  Phoenix       to  Boston
           representing an OBPL
rcvcm 5
  Control message number  5  representing an OBPL  has been received.
note     Now let Phoenix receive the OBPL that was previously sent by
note     Boston.
rcvcm 3
  Control message number  3  representing an OBPL  has been received.
```

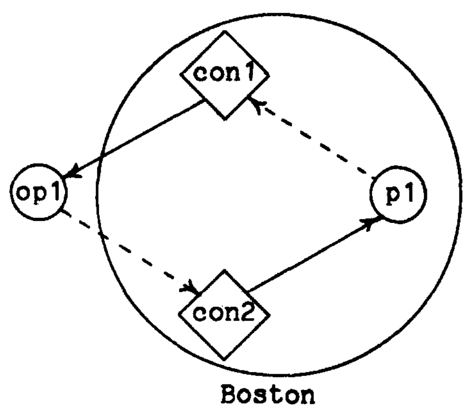


State where control message 3 representing an OBPL has just been sent from Boston to Phoenix. Receipt of the OBPL is delayed until after the state drawn below is reached.



Final State Diagram

```
scenario demo11
note     This is an example of a deadlock involving one process and one
note     operator at the same node.  Two operator connections are involved.
dclop Boston op1
  op1  has been declared as an operator  at node  Boston
copcon con1 Boston op1 p1
  Operator connection  con1  has been established
copcon con2 Boston op1 p1
  Operator connection  con2  has been established
note     Let p1 in Boston request a message from operator op1 in Boston
rcvopmsg con1
  Process  p1             at node  Boston        is blocked waiting for a
           message over operator connection  con1
  An OBPL has been queued waiting for a status  report from operator  op1
           at node  Boston         The involved operator connection is  con1
note     Create a deadlock by reporting that op1 is waiting for a message
note     over operator connection con2.
opstat Boston op1 waiting con2
  We will now check for deadlock involving the given  operator
           and operator connection
  A deadlock has been detected.  The following processes are involved:
                      p1             at node   Boston
                      op1            at node   Boston
           End of deadlock list
```

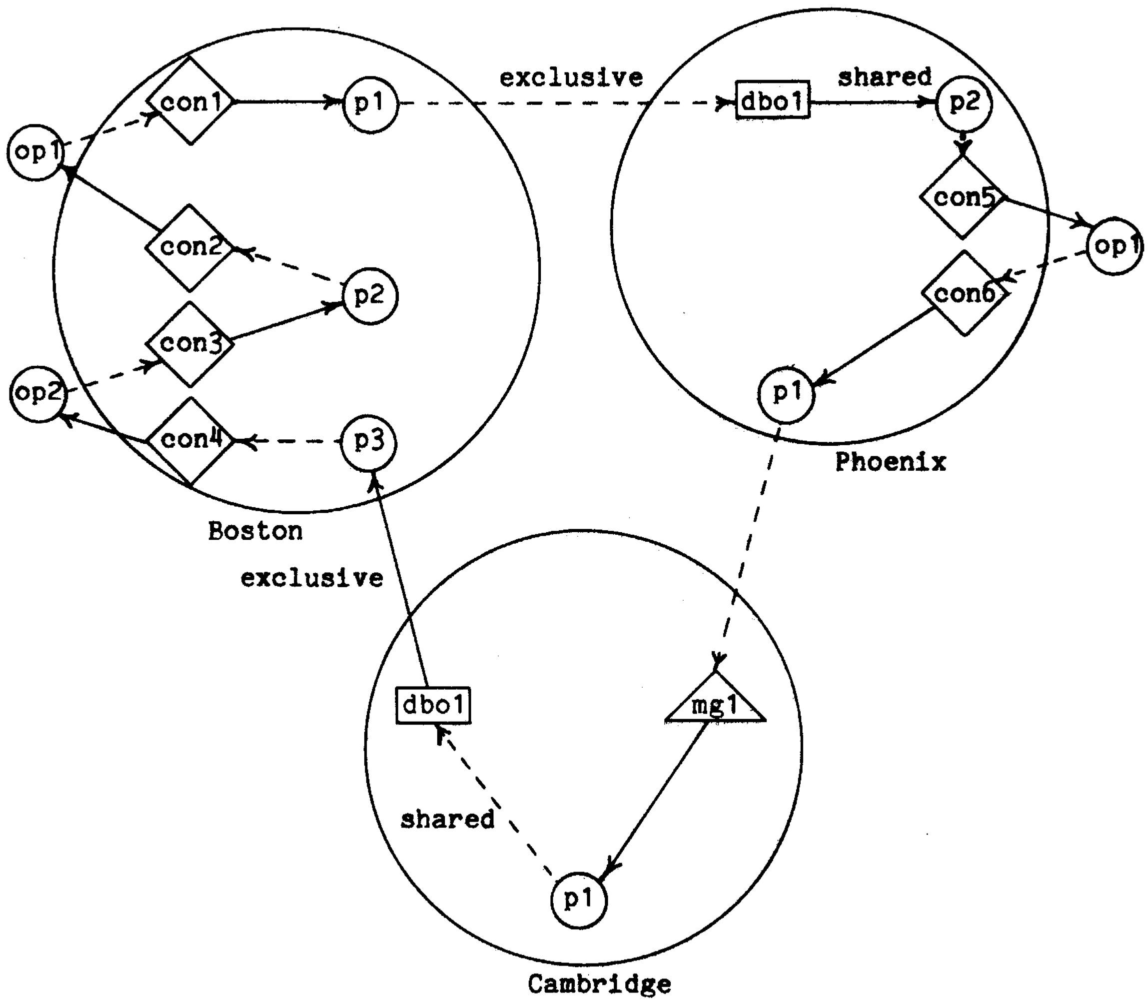


Final State Diagram

```
scenario demo12
note     This is an example of a deadlock across three nodes which involves
note     several operator connections.  It demonstrates that deadlock
note     involving operators will be detected as long as the operator
note     properly states what he is waiting for.  The first 15 commands
note     set up the state where all operators have been declared, all
note     operator connections have been created, the message group has
note     been initiated and accepted, and the involved database objects
note     have been assigned to the proper processes.
dclop Boston op1
  op1  has been declared as an operator  at node  Boston
dclop Phoenix op1
  op1  has been declared as an operator  at node  Phoenix
dclop Boston op2
  op2  has been declared as an operator  at node  Boston
copcon con1 Boston op1 p1
  Operator connection  con1  has been established
copcon con2 Boston op1 p2
  Operator connection  con2  has been established
copcon con3 Boston op2 p2
  Operator connection  con3  has been established
copcon con4 Boston op2 p3
  Operator connection  con4  has been established
copcon con5 Phoenix op1 p2
  Operator connection  con5  has been established
copcon con6 Phoenix op1 p1
  Operator connection  con6  has been established
initmg mg1 Cambridge p1 Phoenix
  Message group   mg1  has been initiated
acceptmg mg1 Phoenix p1
  mg1   has been accepted by   p1   at node   Phoenix
rqdbo exclusive Boston p3 Cambridge dbo1
  Process  p3  at node  Boston  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  1  sent from  Boston  to  Cambridge
           representing a remote resource request
rcvcm 1
  Control message number  1  representing a remote  resource request
           has been received
  p3          at node  Boston         is granted exclusive use of
           dbo1         at node  Cambridge
  Control message number  2  sent from  Cambridge     to  Boston
           representing this allocation
rcvcm 2
  Control message number  2  representing a remote  resource allocation
           has been received
  p3          at node  Boston         has been granted exclusive use of
           dbo1         at node  Cambridge
rqdbo shared Phoenix p2 Phoenix dbo1
  p2  at node  Phoenix  granted shared access to  dbo1  at node  Phoenix
note     Let p1 in Boston wait for exclusive use of dbo1 in Phoenix.  No
note     deadlock will be detected because p2 in Phoenix, which controls
note     dbo1 in Phoenix, is active.
rqdbo exclusive Boston p1 Phoenix dbo1
  Process  p1  at node  Boston  is blocked while a request is sent to
           the node containing the desired resource
  Control message number  3  sent from  Boston  to  Phoenix
           representing a remote resource request
rcvcm 3
  Control message number  3  representing a remote  resource request
           has been received
  Resource is not currently available for  exclusive use, process  p1
           at node  Boston        remains blocked
```

```
note     Let p2 in Phoenix now wait for a message from op1 in Phoenix.
note     We then state that op1 in Phoenix is active, so no OBPL's get
note     expanded further.
rcvopmsg con5
  Process  p2           at node  Phoenix       is blocked waiting for a
           message over operator connection  con5
  An OBPL has been queued waiting for a status  report from operator  op1
           at node  Phoenix          The involved operator connection is  con5
opstat Phoenix op1 active
  All OBPL's waiting for the given state  information have been discarded
note     Let p1 in Phoenix wait for a message from p1 in Cambridge.  No
note     deadlock exists because p1 in Cambridge is active.
rcvmsg mg1
  Process   p1          at node  Phoenix       is blocked waiting for a
           message in message group  mg1
  Control message number  4  sent from  Phoenix        to  Cambridge
           representing an OBPL
rcvcm 4
  Control message number  4  representing an OBPL  has been received.
note     Let p3 in Boston wait for a message from op2 in Boston.  The
note     OBPL created when p3 gets blocked will be discarded when we
note     state that op2 is active.
rcvopmsg con4
  Process  p3           at node  Boston        is blocked waiting for a
           message over operator connection  con4
  An OBPL has been queued waiting for a status  report from operator  op2
           at node  Boston           The involved operator connection is  con4
opstat Boston op2 active
  All OBPL's waiting for the given state  information have been discarded
note     Simultaneously block p1 in Cambridge and p2 in Boston.  Then
note     let Boston receive the OBPL from Cambridge that was created
note     when p1 in Cambridge was blocked.  Before we report the status
note     of op1 in Boston, state that op2 in Boston is waiting for a
note     message from p2 in Boston, thereby queuing a second OBPL for
note     information on the status of op1 in Boston.
rqdbo shared Cambridge p1 Cambridge dbo1
  Resource not available, process blocked.
  Control message number  5  sent from  Cambridge      to  Boston
           representing an OBPL
rcvopmsg con2
  Process  p2           at node  Boston        is blocked waiting for a
           message over operator connection  con2
  An OBPL has been queued waiting for a status  report from operator  op1
           at node  Boston           The involved operator connection is  con2
rcvcm 5
  Control message number  5  representing an OBPL  has been received.
  An OBPL has been queued waiting for a status  report from operator  op2
           at node  Boston           The involved operator connection is  con4
opstat Boston op2 waiting con3
  We will now check for deadlock involving the given  operator
           and operator connection
  An OBPL has been queued waiting for a status  report from operator  op1
           at node  Boston           The involved operator connection is  con2
opstat Boston op1 waiting con1
  We will now check for deadlock involving the given  operator
           and operator connection
  Control message number  6  sent from  Boston         to  Phoenix
           representing an OBPL
  Control message number  7  sent from  Boston         to  Phoenix
           representing an OBPL
note     There were two OBPL's waiting for state information from op1 in
note     Boston, therefore two OBPL's are expanded and sent to Phoenix.
note     Let Phoenix receive and expand both OBPL's, and state that op1
note     in Phoenix is waiting for a message from p1 in Phoenix, thereby
note     closing the deadlock loop.  The deadlock will be detected twice
note     because we had two OBPL's being passed around due to the fact
note     that we blocked two processes simultaneously.
```

```
rcvcm 6
  Control message number  6  representing an OBPL  has been received.
  An OBPL has been queued waiting for a status  report from operator  op1
           at node  Phoenix        The involved operator connection is  con5
rcvcm 7
  Control message number  7  representing an OBPL  has been received.
  An OBPL has been queued waiting for a status  report from operator  op1
           at node  Phoenix        The involved operator connection is  con5
opstat Phoenix op1 waiting con6
  We will now check for deadlock involving the given  operator
           and operator connection
  Control message number  8  sent from  Phoenix       to  Cambridge
           representing an OBPL
  Control message number  9  sent from  Phoenix       to  Cambridge
           representing an OBPL
rcvcm 8
  Control message number  8  representing an OBPL  has been received.
  Control message number  10  sent from  Cambridge     to  Boston
           representing an OBPL
rcvcm 9
  Control message number  9  representing an OBPL  has been received.
  A deadlock has been detected.  The following processes are involved:
                     p1            at node   Cambridge
                     p3            at node   Boston
                     op2           at node   Boston
                     p2            at node   Boston
                     op1           at node   Boston
                     p1            at node   Boston
                     p2            at node   Phoenix
                     op1           at node   Phoenix
                     p1            at node   Phoenix
           End of deadlock list
rcvcm 10
  Control message number  10  representing an OBPL  has been received.
  An OBPL has been queued waiting for a status  report from operator  op2
           at node  Boston         The involved operator connection is  con4
opstat Boston op2 waiting con3
  We will now check for deadlock involving the given  operator
           and operator connection
  A deadlock has been detected.  The following processes are involved:
                     p2            at node   Boston
                     op1           at node   Boston
                     p1            at node   Boston
                     p2            at node   Phoenix
                     op1           at node   Phoenix
                     p1            at node   Phoenix
                     p1            at node   Cambridge
                     p3            at node   Boston
                     op2           at node   Boston
           End of deadlock list
```



Final State Diagram

CS-TR Scanning Project
Document Control Form

Date : 11 / 3 / 95

Report # LCS-TR-185

Each of the following should be identified by a checkmark:
Originating Department:

[ ] Artificial Intellegence Laboratory (AI)
[X] Laboratory for Computer Science (LCS)

Document Type:

[X] Technical Report (TR) [ ] Technical Memo (TM)
[ ] Other: ______

Document Information

Number of pages: 180 (186-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
[ ] Single-sided or
[X] Double-sided

Intended to be printed as :
[ ] Single-sided or
[X] Double-sided

Print type:
[ ] Typewriter [ ] Offset Press [ ] Laser Print
[ ] InkJet Printer [ ] Unknown [ ] Other: ______

Check each if included with document:

[ ] DOD Form [ ] Funding Agent Form [X] Cover Page
[X] Spine [ ] Printers Notes [ ] Photo negatives
[ ] Other: ______

Page Data:

Blank Pages (by page number): ______

Photographs/Tonal Material (by page number): ______

Other (note description/page number):

Description : Page Number:
IMAGE MAP! (1-180) UN#'ED TITLE & ABSTRACT, 3-180
(181-186) SCANCONTROL, COVER, SPINE, TRGT'S (3)

Scanning Agent Signoff:
Date Received: 11 / 3 / 95 Date Scanned: 11 / 22 / 95 Date Returned: 11 / 22 / 95

Scanning Agent Signature: Michael W. Cook

Rev 9/94 DS/LCS Document Control Form cstrform.vsd