A SIMULATOR FOR MESSAGE-BASED DISTRIBUTED SYSTEMS

William F. Quinlivan III

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-87-33 August 1987

Abstract

A broad class of physical systems including queueing, communication, and computer networks can be modelled as a collection of computing nodes, or processes, connected over directed arcs representing communication paths. As a means of executing the algorithm from such a distributed system, a sequential program, named SIM, has been constructed which simulates the execution of the distributed program. This simulation can be used to study the operation of the distributed algorithm, and to obtain its performance data. This report explains the structure and operation of this program, and suggests applications.

A SIMULATOR FOR MESSAGE-BASED DISTRIBUTED SYSTEMS

BY

William Francis Quinlivan III, B.S., B.A.

REPORT
Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT AUSTIN
August, 1981

ACKNOWLEDGEMENTS

I would like to thank Dr. J. Misra and Dr. M. Chandy for their help and advice on this project, and suggestions that improved this report.

TABLE OF CONTENTS

1.0 INTRODUCTION

A broad class of physical systems including queueing, communication, and computer networks can be modelled as a collection of computing nodes, or processes, connected over directed arcs representing communication paths. As a means of executing the algorithm from such a distributed system, I have constructed a sequential program, named SIM, which simulates the execution of the distributed program. The text of this program appears in appendix 1. This simulation can be used to study the operation of the distributed algorithm, and to obtain its performance data. This report explains the structure and operation of this program, and suggests applications.

The execution model of the computing node is based on Hoare's paper, "Communicating Sequential Processes" [HOA78]. In the language model (referred to as CSP) suggested in this paper, processes share no data. Instead, all interaction between processes is via messages passed between a sender and a receiving process, which name each other explicitly.

This model of parallel execution and communication is attractive because of the availability of low-cost processors which can each be required to carry out the computation of a single process. These could be interconnected over communication lines so that the topology of a particular distributed algorithm could be embedded in the processor communication graph. From a program verification point of view, this model has merit because a process literally participates in every transaction that has any opportunity for an outside process to change its data - namely, communications. This simplifies proofs of programs because of the reduced chance of interaction between processes.

In order to study these algorithms, it is generally necessary to either have such an embedding architecture for execution, or to map the problem onto a simpler architecture. Such mapping leads to execution of a parallel algorithm on a single sequential machine. This is the approach taken in this program. SIM was developed and run on the University of Texas' DEC-10 computer, using the programming language, Pascal.

At the current stage of development, no source language processing is supported by SIM; its function is rather the run-time support for programs written in CSP-like languages. The notion of a process in SIM is a Pascal procedure which is a single thread running through local variable initialization, parallel execution with other processes (simulated), algorithm result reporting, and process termination.

In CSP, a process is always in one of three distinct states: executing, waiting for communication, or terminated. Ordinarily, a process alternates between executing, and communicating with other processes. At some point, the process may enter the terminated state after an execution phase. The communication consists of an indefinite wait for communication, followed by an instantaneous passing of a message. Such a message passing is loosely refered to as a message firing, or a port firing. This three state behavior is summarized in the state graph shown in fig 1.1.

```
        |
        |
        V
    +--------+                           +--------+
    |  XQT   |------------------------->|  BLK   |
    | STATE  |<-------------------------| STATE  |
    +--------+     /communicate         +--------+
        |
        |
        V
    +--------+
    |  TRM   |
    | STATE  |
    +--------+
```

fig 1.1

The SIM implementation uses a central <u>CONTROLLER</u> which resumes the processes in the network, after they communicate, to execute their own part of the distributed algorithm. In the SIM implementation, the role of the process is played by a Pascal procedure which contains the procedural description of the process it emulates. Hence, the action of resuming a process after communication is accomplished by calling the procedure which implements the process.

A central network-time is maintained which orders the execution and communication of the network, and permits collection of global performance statistics. As long as the individual processes never refer to this global network time, it remains a purely auxiliary variable, serving only to meter the time lost to processes while communicating, or waiting to communicate. Since one definition of a distributed system is one in which communication introduces non-negligible time delays, [LAM78] it seems beneficial to model such delays in a network simulation.

Hoare's paper made no mention of time dependent behavior, except to suggest that, as a fairness issue, process pairs awaiting communication with each other should not be delayed indefinitely often; this is appropriate for a paper making a preliminary language definition. However, to obtain any kind of performance statistics for an algorithm written in such a language, it is necessary to place bounds on wait times incurred by process pairs which are waiting to communicate with each other. In SIM, this bound is synonymous with the "time unit" in which time periods are measured. This is equivalent to saying that communication begins in the first time unit following the event that both partners of a communication pair become ready to communicate. The performance statistics derived from such an approach therefore represent best-case times, which could only be realized in a physical system that similarly bounded mutual-wait times.

2.0 THE PROCESS

The process plays a central role in both CSP and SIM. The nature of the process in both CSP and SIM, and the mapping of CSP processes into the "process-procedures" of SIM is the subject of this section.

2.1 The Process in CSP

In the source text, a process in CSP consists of three items:

. a name, which can either be a simple identifier, or an identifier followed by index limits, specifying an array of similar processes.

. a local variable declaration part which defines the local data structures and specifies initial values.

. a procedural description of the actions of the process. This program is specified in terms of six control structures: guarded commands, assignment commands, parallel commands, repetitive commands, alternative commands, and the I/O commands that Hoare maintains are primitive.

The reader is referred to the original CSP paper for a complete description of the process textual structure, and the semantics and a recommended syntax for the command types, together with illustrative examples. An important characteristic of any I/O command that appears in the text of a process, P, is that it specifies the identities of the processes with which P will be eligible to communicate, when the command is encountered during P's execution.

The CSP model of a process is a three state finite state machine, (FSM) which makes some number of transitions between executing and waiting for communication, followed optionally by a transition from the executing to the terminated state. Communication is treated as if it occurred instantaneously after an indeterminate period of waiting.

The entire address space of the process is local to the process. The process communicates with other processes by naming them explicitly in an input or output (I/O) command, and subsequently having that communication selected for firing by scheduler whose selection policy is arbitrary. The process may wait simultaneously for communication with many other processes, but the scheduler never chooses more than one message for firing per process. As part of the communication, the parties involved are informed of the identity of the processes with whom they communicated.

When a message fires, the effect is the same as an assignment statement where the expression yielding the value assigned is evaluated in the sending process, and selection of the receiving variable is performed in the receiving process. As in an assignment statement in a strongly typed language, the type of the value sent must match the type of the receiving variable.

Although the process is never involved in communication with more than one other process at a time, parallelism is introduced in the I/O handling because a process may wait for communication with more than one process at a time. This arises in two ways. The first is that a single I/O statement may name an entire array of processes as eligible for communication, and the second is that the process tries to execute some I/O commands in parallel. Actually, only one of the commands will be selected for execution. Three separate statement types give rise to this latter type of parallelism: the parallel command, the repetitive command, and the alternative command.

The repetitive and alternative commands are borrowed from similar commands suggested by Dijkstra in [DIJ77]. The repetitive command:

```
* [  P1 ? y --> x1 := x1 + 1 ;
   □ P2 ? y --> x2 := x2 + 1   ]
```

causes the process to wait for input from either P1 or P2, and depending on which one does send a message, either x1 or x2 is incremented, and the statement repeats. Here, only one message fires at a time, but the process always waits in parallel. The alternative command:

```
[ P1 ? y --> x1 := x1 + 1 ;
□ P2 ? y --> x2 := x2 + 1   ]
```

is similar to the repetitive command except that once either process sends a value for y, the respective increment of x1 or x2 is performed, and the statement terminates. The parallel command:

```
[ P1 ! x || P2 ! x ]
```

forces the containing process to wait to send the value of x to both P1 and P2 in any order. Hence, before either message fires, the process is waiting in parallel.

This then is the kernel of process functionality that is required in an implementation: the basic structure and execution behavior of the process as described above, including parallel waiting for communication. Hoare has further requirements such as termination of a repetitive command when all of the named-for-communication processes have terminated, but this is not supported in the implementation. Hoare also disallowed the presence of output statements (i.e. message sending) in the guards of the repetitive and alternative commands, but the SIM program does not make this restriction. SIM can do this because the states of all processes are available in a single memory to which the scheduler has instantaneous access.

2.2 The Process in SIM

In SIM, the process appears as a Pascal procedure describing not a named process or an array of such processes, but rather a generic process type. These procedures are all compiled and mapped with the SIM code. In a particular execution of SIM, a given process type may be instantiated zero, one, or more times. Most SIM runs will exercise instances of more than one process kind. All process instantiations are assigned a unique identifier, known locally as ID, that is an input parameter in the calls to the procedure containing the process definition. Such calls correspond to a resumption of the process being modelled, and are typically handled by means of a call to the RESUME procedure.

All of the local data of all the processes is contained in the array OWN. This array is indexed by the values of the process identifiers. In an actual distributed system, this would be all of the state information necessary. However, since SIM uses a central scheduler, information on the process' current state is kept in a separate array LPS, also indexed by the process identifiers. The structure of the LPS entries is the same for all of the processes, but the OWN entry for a process is determined by the kind of process that owns it.

In SIM, communication lines are known as ports, and each port has a fixed sender, receiver, and type of message. Since it would be very restrictive to require the processes to know the identity of all the other processes at compile time, unique port names, instead of process names are given in the I/O commands which give rise to communication. The unique port names exist as the values of local variables in a process. Hence, a process may have an output command which specifies a variable OUTPORT as the port over which a message should be sent. The particular port over which this command specified message passing would depend on the value of this variable at a given moment.

During the initial phase of a SIM execution, all such local port names are bound to particular ports; also, the sending and receiving process identifiers, and the message type allowed are bound to the particular port. This latter information is kept in the array PORTS, which is indexed by the port identifiers. If all such binding is made consistent with some communication graph, G, then the topology of G is reflected in the connectivity of the processes and ports. This scheme effectively isolates the procedural specification of the process from the particular topology of the network to be simulated. In other words, the process semantics, but not the network topology is bound at compile time; the topology is specified at run time, and may vary between executions of SIM. Use of port names instead of process names was also suggested by Hoare in the original paper.

The actions of the process may be viewed as responding to a set of significant events by deciding what the next event should be, and updating its local variables. The events here are the initial invocation of the process, the sending of a particular output, and the receiving of a particular input. In SIM, each event is associated with a label preceeding the code which responds to the event. During any particular execution, the process specifies the set of next events, and then returns to the central CONTROLLER. To the process, this is equivalent to invoking an "oracle" which selects one of the events and resumes the process at the label associated with that event.

The labels then are the points at which the process may potentially be resumed. As such, they may be viewed as the addresses of a program of the form:

ADDRESS	ACTION
label1	respond to event corresponding to label1
label2	respond to event corresponding to label2
.	.
.	.

A variable that ranges over these labels is known as a meta-program counter, abbreviated MPC. Each resumption of a process gives as one of the parameters the particular MPC value at which execution should continue.

A special procedure, named PARWAIT (for parallel wait) is provided for the purpose of telling the CONTROLLER about the events for which the process will next wait. To wait for communication over a set of ports, one PARWAIT call is made for each port. If a process returns to the controller without invoking PARWAIT, it is marked "terminated", and dropped from further simulation.

PARWAIT has three parameters: the port name over which communication may take place, the identity of the requesting process, and the MPC value at which control should resume if the communication described in the current PARWAIT invocation turns out to be the next communication that the calling process is involved in.

Several of the MPC values are reserved for events which are common to all of the process kinds. MPC = 0 selects code activated at process creation. Here local variables are initialized which do not change throughout the entire life of the process. These include things like the local variables that name ports, and parameterizing variables, such as the buffer size of a bounded queue, or the delay parameters for a process which simulates a time delay process. MPC = 1 corresponds to initial process activation. Here local variables are given values which may change during execution. Activation and creation have been kept separate to allow several activations of a process that only needs to be created once. During the activation, the process decides and informs the CONTROLLER which events will be the first ones allowed by the process. MPC = 1000 is reserved for code which allows reporting of results after the simulation has terminated, and do things like close output files if this is necessary. This reporting is strictly of things known locally; the overall performance reporting for the entire network is handled by the central CONTROLLER. Two other MPC values, 1010, and 1020 have dedicated functions relating to a deadlock recovery technique discussed in section 4.3.

All other MPC values are provided to specify the code to respond to events. The only other restriction on these MPC values is that they be unique within a particular process.

Each process instance has a message buffer that is used for communication. This buffer is part of the process' activation record in the LPS array. When process I wants to send a message over port X, where MPC = Z corresponds to the code segment that responds to this communication, it first writes the message into its message buffer, and then invokes PARWAIT with the appropriate parameters: port = X, requester = I, and MPC = Z.

If process I were to be the receiver, instead of the sender of this message, the message is not written into the buffer (since I does not yet know its contents), but the PARWAIT call is the same. When I is resumed after the message is passed, it will find the message in its buffer.

Thus, processes are supported with all of the necessary functionality. As long as a user of this program were content to code the process descriptions in the necessary form for execution, SIM would be usable as described. Typically, however, the user will wish to write the process code in a language that allows concentration on the semantics of the process, and not worry about MPC values and PARWAIT parameters. The next section gives the outline of a translation procedure to convert the CSP commands into code compatible with SIM.

2.3 Mapping CSP Programs into SIM Programs

The process of converting a program written in CSP into the equivalent program suitable for execution on SIM consists of translation of the main program, and all of the individual processes.

2.3.1 Mapping a CSP Parallel Program Segment

The typical form of a parallel program segment in CSP is:

```
[ P1 || p2 || ... || Pn ]
```

where the P1...Pn are the processes defined in the program. This just says that they all should run in parallel. This structure is the implicit execution model for a SIM main program. Main program structures that depart from this form require the definition of a special process that performs the function desired in the main program.

Such a "main program process" would run just like the other processes, but would be the only one that performed any meaningful computing before becoming blocked. The normal call-return structure familiar in sequential programs is easily implemented by having the processes pass special "call", and "return" messages, where the content of the message is made up of calling and return parameters, respectively.

2.3.2 Mapping CSP Processes to SIM Process-Procedures

In order to convert a CSP process into the equivalent SIM process-procedure, it is necessary to map the three parts of the CSP process. These are the name, the local variable declaration, and the programming language statements that state the execution behavior of the process.

The name conversion consists of placing the process name into the scalar list defining the type PROCKIND, and into the PRPROCKIND procedure that prints out the name of a process in SIM. The name must also appear in the RESUME procedure that invokes a process after it communicates.

Mapping the local variable declaration section means placing the structural definition into the definition of the data type, OWNDATA. The structure of a variable of this type is determined by the kind of process that owns it; this information is kept in the field <u>OPROCKIND</u>. In general, many data templates will appear in the definition of <u>OWNDATA</u>, but the inclusion of a particular local data type is straightforward, as can be verified from the example in the appendix. This inclusion makes the definition needed by the process. To implement the initialization of this data, the code must appear in the process initialization part of the definition of the process. By convention, this function occurs at <u>MPC</u> = 1.

An implicit data type exists in the form of the messages passed by the process. In CSP, this is handled by insisting that the type of all messages match the type of the variables which receive their content. This is not supported in SIM. The messages passed by a process must have a defined structure. The structure for all message types is made in the definition of the SIM data type, <u>MESSAGE</u>. The actual structure of a message is user defined, and a particular network simulation may contain messages of more than one type, e.g. DATA and ACKNOWLEDGE. In this case, this type has a variable structure determined by the value of its <u>MSGKIND</u> field. Hence all message kinds in a simulation must have their name listed in the definition of the scalar type, <u>MSGKIND</u>, and the corresponding message structure must appear in the <u>MESSAGE</u> definition.

The information about the graph topology that is explicit in a CSP process is handled in two stages in SIM. At compile time, the ports are assigned local variable names in the processes. Then, at network specification time during the actual running of the program, the values of these variables are bound to specific ports in the network.

The translation of the parts of the process described above is fairly mechanical. This leaves the procedural description of the actions of the CSP process as the major challenge in converting a CSP process into its SIM equivalent.

The only parallelism or nondeterminism supported by SIM is the parallel waiting for communication. This is "parallel" because the process waits for more than one process at a time, even though the waiting will be terminated whenever any one of the named ports does actually fire. The nondeterministic element is introduced because the process does not know in advance which of its potential communications will be the one to fire.

CSP allows other forms of local parallelism in commands like:

```
[ x := x + 1 || y := y + 1 ]    ...and...
[ guard1 --> x := x + 1 ;
□ guard2  -->  y := y +1     ]
```

where both guards are true. An implementation for this requires a local scheduler to evaluate guards and select eligible actions. This low-level nondeterminism is not supported by SIM, so a translator would have to generate code to arbitrarily select one of the assignments in the first example to perform, and to select one of the guards in the second to evaluate first, and if found to be true, perform the indicated action.

Hence, the main difficulty in translating CSP processes into equivalent SIM Pascal procedures is to translate the parallel, repetitive and alternative commands in such a way that the parallel waiting for communication is preserved. These will be discussed in turn.

The general form of the repetitive command is:

```
* [ <boolean expression 1> ; <I/O command 1> -->
                 <action 1> ;
  □ <boolean expression 2> ; <I/O command 2> -->
                 <action 2> ;
          .                              .
          .                              .
  □<boolean expression n> ; <I/O command n> -->
                 <action n>                              ]
```

Here, the process will wait for any of the I/O commands which have the corresponding boolean expression true. Once one of them fires, perform the corresponding action. Since the set of awaited lines must be specified when the process returns to the controller, it is necessary for the process to evaluate all of the boolean expressions, and request parallel waiting for the corresponding I/O command for each one that is true. This also requires that a condition, once evaluated to be true, must not be subsequently falsified, e.g. through a side effect of evaluating a subsequent expression. Consider DECIDENEXT to be a shorthand notation for the Pascal statements:

```
     if not (<boolean expression 1>  or
                   <boolean expression 2>  or
                              .

                              .
                   <boolean expression n>)
                      then go to L1
      else for i := 1 to n do
           if      <boolean condition i>
                      then PARWAIT(porti,id,mpci);
```

Here L1 is the label of the next statement after the statements associated with this repetitive command, and porti is the port named in the i-th I/O command, and mpci is the MPC label associated with the code which should follow I/O command i. The equivalent SIM coding for this command is:

```
          DECIDENEXT;
mpc1:     begin <action 1> ; DECIDENEXT ; end;
mpc2:     begin <action 2> ; DECIDENEXT ; end;
  .              .                       .
  .              .                       .
mpcn:     begin <action n> ; DECIDENEXT ; end;
L1  :     <rest of program> ;
```

With this interpretation, the command would terminate when all of the boolean conditions were false.

The general form of the alternative command is:

```
[ <boolean expression 1> ; <I/O command 1> -->
          <action 1> ;
□ <boolean expression 2> ; <I/O command 2> -->
          <action 2> ;
      .                          .
      .                          .
□ <boolean expression n> ; <I/O command n> -->
          <action n>                         ]
```

This CSP command is supposed to begin waiting in parallel for all I/O statements with a true guard. If none are true, the statement should fail. Unlike the repetitive command, the action is only performed once. SIM has no feature corresponding to having a statement fail. Hence, if it is desirable to have the process be terminated in response to execution of this statement with all guards false, then the DECIDENEXT procedure defined above should simply return to the central CONTROLLER instead of the statement fragment:

```
go to L
```

Since the process will have returned without specifying any I/O, the CONTROLLER will mark the process as terminated. Otherwise, the DECIDENEXT can be used as is, and the semantics of the repetitive command become: if there is no true boolean condition, then just go on to the next statement, otherwise, begin waiting in parallel for an I/O statement with a corresponding true guard; after one of them has fired, execute the associated action.

With this possible modification to the DECIDENEXT procedure, the code for a SIM interpretation of the repetitive statement becomes:

```
            DECIDENEXT;
mpc1:       begin <action 1> ; go to L1 ; end;
mpc2:       begin <action 2> ; go to L1 ; end;
  .                 .                   .
  .                 .                   .
mpcn:       begin <action n> ; go to L1 ; end;
L1  :       < rest of program  >
```

The general form of a parallel command that may result in parallel waiting for I/O is:

```
[  <I/O command 1> || <I/O command 2> || ...
                                   ... || <I/O command n> ]
```

A possible means of handling this is to establish an n-element boolean array, DONE, and convert this parallel command to the equivalent CSP statements:

```
{ set all DONE[i] to false }
* [ DONE[i] --> DONE[i] := false ]
* [ not DONE[1] ; <I/O command 1> --> DONE[1] := TRUE;
    not DONE[2] ; <I/O command 2> --> DONE[2] := TRUE;
          .              .              .           .
          .              .              .           .
    not DONE[n] ; <I/O command n> --> DONE[n] := TRUE; ]
```

The resulting repetitive command would then be translated according to the rules described above into the equivalent SIM statements.

3.0 THE IMPLEMENTATION OF SIM

This chapter will describe in considerable detail the data structures, flow-of-control, termination conditions, statistics collected, and debugging aids of the SIM program.

3.1 SIM Data Structures

SIM allows execution of networks consisting of several varieties, or types, of processes. To qualify the kind of particular processes, a scalar type, PROCKIND, is defined which ranges over all possible process kinds. The current execution state of a process is contained in entries of two arrays which have already been mentioned. These are the OWN and LPS arrays, which contain the local variables and activation records, respectively, of the processes. Both of these arrays are indexed by the unique process identifiers.

The element of the LPS array is of a structured record type ACTREC which contains the following information:

. the type and instance of the process. Together, these two fields uniquely identify the process in a way that has mnemonic value to the user. As an example, they could specify a [server,6] or [queue,2] in a network where process types server, and queue are supported.

. the current state and the next state of the process. The current state is one of executing (XQT), blocked (BLK), communicating (CMN), or terminated (TRM). The next state can only be BLK or TRM, and is used for a process to inform the central CONTROLLER of the state it will enter following its current phase of execution.

. the time-left field is used as a count down timer. When this reaches zero, it signals the end of the current state, either CMN or XQT. Initial values for this timer come from the process itself for execution time, and from the port records for communication delays (see description of PORTS, below).

. three accumulators record the total process time spent in the states XQT, CMN, and BLK. In case a process has terminated, a separate field contains the network time at which this occurred.

. one buffer each is provided to hold an MPC, a port identifier, and a message. These are used in communicating the actual message passed, the MPC value at which a process should resume, and the identity of the port over which a message has just been passed, when the central CONTROLLER resumes the process after a message firing.

In short, the LPS entries contain everything the CONTROLLER needs to know to change the state of the process, handle the details of message passing, and collect process-specific statistics on its performance.

The OWN array also has elements of a structured type, called OWNDATA. This record type contains a field of the PROCKIND type which tells the type of the process with process identifier equal to this element's index in the array. The rest of the OWNDATA structure is variable, and depends only on the kind of process that owns it. Typically, this contains the local variables that name ports incident on the process which are referenced in I/O statements. This may also contain a local variable for local time, if this is desired, and and any data for holding locally-maintained statistics, such as queue length information that might be kept by a process implementing a queue. Basically, this array must hold all of the process' data which must survive between calls to the procedure that implements the process. Strictly temporary variables can be handled by the regular Pascal local variables.

Since all of the field names of Pascal record types with variable fields must be unique, some care is required to prevent collisions in the name space of the variants for different process kinds. A practical solution which has been exploited in the example SIM program in the appendix, is to prefix all field names with a two- or three-letter combination which is suggestive of, and unique to the owning process kind.

Port identifiers in the simulated network are also unique, and serve to index the PORTS array. PORTS[j] contains all of the information held about the port with unique identifier, j. The element type of this array, called PORTREC, contains the following information:

- the unique process identifier of the sending and receiving process, which remains constant throughout an execution of SIM.
- four boolean fields tell whether the sender and receiver are ready and/or eligible for communication. Eligible means the process named this port for communication during its last execution phase by means of PARWAIT calls, and ready means that the process is currently blocked. Actual message passing on this port will never take place unless the sender and receiver are both ready and eligible.
- the MPC values of both the sender and receiver which should be returned to when and if this port fires. Since the processes may wait on many messages, each with a separate MPC return point, a separate MPC must be kept for every possible communication.
- the amount of time that communication over this port delays both the sender and receiver. The interesting cases seem to be when the send-time is less than or equal to the receive time. The send time is related to the size of the message, and the communication rate. The difference between send and receive time corresponds to communication delay. Hence message propagation delays can be explicitly simulated,

e.g. in a simulation of communication over satellite links.

. a count of the total number of messages that have been sent over this port since the beginning of the run.

. a buffer that holds the message, if any, that was specified most recently by the sender process, just prior to the PARWAIT call which marked the sender as eligible for communication over this port. The MESSAGE record type, contains in addition to the actual message, a message type. The type in the message associated with this port binds the port to a single message type. Attempts by a process to send a message with a type different from that recorded in the port's message buffer results in an error message.

The global variables maintained by the central controller include the global time, accumulators for time spent by processes in each of the states XQT, BLK, TRM, and CMN, and a count of the total number of messages sent. Several trace variables are used to turn the run-time trace off and on. This trace is useful for debugging both the process procedures and the SIM program itself, and will be discussed below. Other variables keep termination thresholds for time and message counts, and deadlock occurrences.

3.2 Flow of Control in SIM

The sequence of events in a run of SIM consists of the following steps:

. an interactive session which solicits inputs from the operator about the connectivity of the network to be simulated. This information could optionally be read from a file. This session obtains information in two categories:

- bind each unique process identifier to a particular process kind and instance by instantiating the defined process kinds zero, one, or more times.
- for each of the ports in the network, bind the sender and receiver processes; this implicitly specifies the topology of the network.
- simulate the execution of the network, collecting statistics as progress is made.
- print the statistics gathered in the last step.

The execution model of SIM during the simulation of the target network is the familiar operating systems concept of multiprogramming, where a number of tasks (processes) that are not blocked (whose state is XQT) are each allocated one time quantum of compute time in round-robin fashion. At the limit where the quantum size becomes zero, this becomes an instance of processor sharing, and an external observer sees all executing processes making steady progress. During each time unit in a true distributed system, the volume of processing performed is the product of one time unit and the number of processes that are executing at that time. This of course assumes that all processes that are executing accomplish the same amount of processing in a time unit. Hence, it seems reasonable to asert that one time unit of simulated time has passed every time that the central CONTROLLER has made one pass through the list of executable processes; equivalently, this grants one time quantum to each ready process each time the network time is incremented.

An important point here is that the process is not actually resumed during every time unit in which it is in the XQT state. Processes are only resumed when they make a transition into the XQT state. The actual amount of time required for the process to complete its actual computation is immaterial. All that matters is that the process inform the CONTROLLER of the simulated time required to complete the execution being performed. This is used for performance purposes only. In cases where only the execution of an algorithm, as opposed to its simulation, is important, this time value can be ignored altogether.

Hence, a process simulating a time delay of 25 time units need only tell the CONTROLLER that it will require 25 time units. The time required to return this result does not matter. The effect of resumption, from the CONTROLLER'S point of view, is that an oracle is invoked which provides this time estimate, and the set of ports over which this process will be waiting to communicate during the block state which will follow the current execution state. In the case that no ports will be awaited, the action simulated is that the process is getting ready to terminate. Furthermore, the oracle can provide the contents of any message that the process will wait to send. Lastly, the oracle can assert that the process has updated its local variables as if the execution had already taken place, so for the rest of the simulated time period, nothing at all needs to be computed. These are exactly the effects of resuming a process, and as a result, the CONTROLLER can merely simulate the passage of time rather than actually giving a time quantum to the process every time the network time is incremented.

The estimate of the total compute time is returned by the processes in the XQTTIME field of their LPS entry. The central CONTROLLER copies this value into the TIMELEFT field; at every subsequent time unit, this value is decremented until it reaches zero. At this point, the process enters the next state, always one of terminated, or waiting for communication.

Similarly, any process in the CMN state has its time left field decremented, and the process reenters the XQT state as soon as the count reaches zero. Here the total time is provided not by the process, but rather by the user during system specification early in a run of the program, who presumably knows the intrinsic characteristics of the communication port, and the size of the messages that pass over the port.

Processes in the BLK state have no such bound on the amount of time they will remain in this state. They simply wait until one or more of the processes that they are wating to communicate with become ready for message passing. If a process is found to be waiting only for processes that have terminated, it is marked terminated.

The above mentioned processing takes place in the procedure TICK, which charges each process according to its state, and initiates those state changes which occur at predictable times. This routine is called once each time unit.

The procedure PASSMESSAGES is also called once every time unit, and its action is to fire some of the ports which have both sender and receiver both ready and eligible for communication over the same port. The current implementation uses a fair scheduler which guarantees that no such ready-to-fire port will be passed over more times than there are ports. This is accomplished by keeping a variable, called netfair which ranges over all the port identifiers, favoring them for a message pass if their sender and receiver are prepared to communicate, and incrementing to the next port at each time unit.

The scheduler is also deterministic in the sense that if two processes are both ready and eligible for a message pass, over say port J, then at least one of them will be involved in some message passing, either over port J, or over one of the other ports that is incident on the process. The only reason port J would not be selected for firing is that the scheduler already selected a different port incident on the port J sender or receiver. In this case, port J could not be fired because that would mean some process was involved in more than one message firing.

The scheduler has been coded as a separate procedure so that a user may provide a different algorithm for selecting among the ready ports to decide which, if any, of them should actually fire.

When a message does fire, the contents which have been buffered in the port's records are copied to the receiver process' message buffer. Both the sender and receiver are placed into the CMN state to wait out the simulated communication time. This takes place in the procedure, FIREPORT.

The simulation view of the processes is now complete. Compare the state graph for the SIM process-procedures in fig 3.2 with the state graph that describes only the logical view of a process' states, fig 1.1. The nodes correspond to the states, and the arcs are the transitions. The procedures named for the arcs are those invoked to make the transition for the processes. Not shown in the state graph is a self loop from each of the states to itself, taken implicitly in the TICK procedure. The lengths of time spent in each state are determined as follows:

. time in the CMN state is determined by the properties of the port handling the message, and the length of the message. This information is held in the PORTS array.

. time spent in the BLK state is a function of the readiness-to-communicate of the processes named for potential communication by a process.

. time spent in the XQT state is determined by the process itself.

. time spent in the TRM state is from the time the state is entered until the end of the simulation.

```
          +--------+         (FIREPORT)           +--------+
          |  CMN   |<-----------------------------|  BLK   |
          | STATE  |                              | STATE  |
          +--------+                  +---------->+--------+
              |                       |               |
              |                       |               |
   (RESUME)   |                       |               |  (AXE)
              |                       |               |
              |                       |               |
              V                       |               V
          +--------+   (TICK)         |           +--------+
 (RESUME) |  XQT   |------------------+           |  TRM   |
--------->| STATE  |--------------------------------->| STATE  |
          +--------+         (TICK)               +--------+
```

fig 3.2

3.3 Statistics collected by SIM

As part of the activation record of a process, SIM keeps a running sum of the total amount of time that the process is in each of its states, and the network time at which the process terminated, if this has occurred. For each port, SIM records the total number of messages passed along the port. SIM also keeps the total elapsed network time.

The procedure PRINTSTATISTICS is used at the end of a run to print out the statistics collected during the run. This routine can be extended to also print out any other statistics which may be deemed necessary for implementation of a target network.

It is appropriate for the processes themselves to keep track of some types of performance statistics, and to print out this information at the end of the simulation. For this latter purpose, the MPC label 1000 is provided. Processes are resumed at this label one last time when the simulation is complete for the purpose of printing this data. As an example, queue length distribution histograms could be printed out by processes implementing a waiting queue.

3.4 Debugging Considerations and Simulation Termination

In order to facilitate implementation of SIM and the various target networks that may be developed using this program, provisions have been made to force SIM to produce various amounts of auxiliary, or trace, information while it is simulating the network. This output is controlled by the values of seven "trace" variables which the user is able to change at critical points during the run. Each of these variables controls the output within a particular area. The output produced is none if the variables are set to zero, and generally more output is produced the larger the variables are assigned. The trace variable name and area of execution trace controlled by the variable is summarized in the following table:

Variable	Trace Area Controlled
SIMTRACE	General execution of SIM
NETXQTTRACE	Processes whose state is XQT
NETBLKTRACE	Processes whose state is BLK
NETCMNTRACE	Processes whose state is CMN
NETTRMTRACE	Processes whose state is TRM
NETDEADTRACE	Network deadlocks
TARGETTRACE	Operation of target network

Discussion of network deadlocks is delayed until section four.

During SIM execution, the user provides execution parameters in response to specific questions presented by the program. Two of these are the time and message limit. When either of these user-supplied limits is exceeded, rather than automatically terminating, SIM gives the operator a chance to set new limits in order to continue the simulation. The program will terminate immediately if the operator does not increase the limit which was exceeded.

The time and message limits, and the trace variables mentioned above are all solicited together by a procedure named SETTRACE. Accordingly, by judicious choice of the limits and the trace variables, the operator can select particular trace options between particular times or message counts. For example, if a target network has been observed to blow up near network time = 100, then the operator may leave the trace values low between times zero through 95, and then, when the time limit of 95 is exceeded, increase the time limit to say 105, and increase the trace variables so that a large volume of trace information about the target network will be printed out between the current time and the new time limit.

This is the exact use intended for the trace variable TARGETTRACE. Of course, since much of the implementation of the output from the target networks is necessarily handled by the processes themselves, it is necessary for the code to be included in the processes so that a higher value of TARGETTRACE does in fact produce more output from the network. This is evident in the example processes in the SIM listing in the appendix.

As an aid in including such code in the individual processes, P, of a new type, the following routines are provided:

Routine	Prints Out
PRSIGNATURE(P)	P's identity, kind and instance
SHOWOWN(P)	P's local variables
SHOWPROCESS(P)	P's activation records from LPS
SHOWMSG(M)	Message M's contents
SHOWNETWORK	Port - Process connectivity graph
TRACELP(P)	Useful process information
DUMPTTY	All global data, PORTS, LPS. etc.

4.0 AN APPLICATION -- DISTRIBUTED SIMULATION

Distributed Simulation [CHA81] has been proposed as a fruitful applications area for computer architectures exhibiting parallel processing capabilities. Here, the parallelism in a computation mirrors the parallelism inherent in a set of physical entities to be modelled. Interactions between the entities are simulated by messages passed between the processes that simulate the entities.

This is an instance of a message-based distributed program that is amenable to simulation on the SIM program. Here, the communication lines are modelled as ports, and the roles of the processing nodes are filled by SIM process-procedures.

In the paper presenting this concept, Chandy and Misra define the notions of process-times and line times. These are the times in the entity-level through which a process or port has been simulated. A key correctness requirement is that the events in a process or a port must be totally ordered in time, even where the time is in a "logical" sense, i.e. measured by a counter, as opposed to time measured by a clock. A port exhibiting this property is said to be "chronological". Lamport [LAM78] also treats the subject of local and global clocks, and describes the need for an extension of the partial ordering imposed by the collection of individual local clocks, to a total ordering of the events within the network if they are to be executed in a consistent manner. This is exactly the function of SIM.

In the physical system being simulated, the analog of line and process times are all trivially equal because time is always constant across the entire network. This is not so at the logical level of the simulation, and in general processes and ports may have radically different local times. This corresponds to the situation where some parts of the simulation are running far ahead of others, and the possibility of this asynchrony is the motivation for Distributed Simulation in the first place. The asynchrony represents parallelism in the simulation that is not possible in the entity level network.

In order to be able to simulate an interesting set of networks, it is necessary to define the individual process types which will comprise these networks. The next section will define six different types and explain the semantics and waiting rules for each of the types.

4.1 Some Process Types for Distributed Simulation

A very rich class of networks can be simulated by using a small set of process types. The processes constructed for this application are as follows:

- the SOURCE type which generates "jobs"
- the SINK type which consumes jobs
- a DELAY type which makes a job wait for some time
- a FORK2 type which accepts jobs on an input line, makes a decision, and sends the job out over whichever of its two output lines is selected by the decision.
- a MERGE2 accepts jobs over two input lines and sends them out over a single output port while guaranteeing the chronological condition on the output line times
- a QUEUE20 models a waiting queue with a maximum size of 20 jobs.

In the following sections, each of these process types will be explained. The QUEUE20 process type is described in greater detail so that the relationship between its name, message types, local variable structure, procedural definition, and the facilities of SIM can be better understood.

4.1.1 The SOURCE Process Type

A process of type SOURCE is connected to the rest of the network via a single port, known to the process as the value of the local variable, <u>SOOUTPORT</u>. When the process is created, i.e. resumed at <u>MPC</u> = 0, this variable is bound to one of the ports in the network being simulated.

A SOURCE type process sends messages out over its output port with an interdeparture time determined by the code implementing the process. One application could read the departure times from a file. This might be used to perform trace-driven simulations, where the input to a simulation was a set of actual values measured in a real physical system. The SOURCE process listed in the appendix uses as the interdeparture time the sum of a constant, contained in the local variable SOCON, and a "discretized" exponential random variable, with parameter SOMU generated using the RANDOM function available on the DEC-10.

When the process is created during a run, the operator supplies values for the variables SOMU and SOCON, and binds the output port to a specific port in the target network by specifying its unique port identifier as the value of the variable SOOUTPORT. Such a SOURCE "node" also keeps a local time, in the local variable, SOTIME, which is always the time that the last message was sent out. The mechanism for sending a message after the passage of some delay, say 10 time units, is to create a new message with a timestamp of SOTIME + 10, and place the message into the process' message buffer. This value also replaces the current value of SOTIME. Then the interdeparture time, 10, is also used as the XQTTIME returned to the central CONTROLLER. This tells the controller that the process will be executing for 10 time units before it tries to send out any message over its port.

Finally, PARWAIT is invoked to indicate that the process will eventually wait to send a message over the port whose identifier is the value of SOOUTPORT.

After this port does fire, this simple program is merely repeated; each time a positive delay is computed, and used as the time lapse until the next message will depart. Since the successive message times are increasing, the chronological condition is met.

4.1.2 The SINK Process Type

The SINK process type has a single input port, known locally as SIINPORT. The SINK process always waits for input, and when a message is received, it is disposed of. In the current implementation, nothing is done with this message. Other uses of the SINK process might have the SINK process record some data from the message, or possibly print out a message as it is received. A SINK has no output ports, and so trivially guarantees monotonicity on its output ports.

4.1.3 The DELAY Process Type

A DELAY process waits initially for input over an input port, known as DINPORT, and after a message is received, computes a time delay in the same way as the process type SOURCE. After this time delay is waited out, the process waits to send out the same message over a port named DOUTPORT. Waiting out the delay is accomplished by leaving the process in the XQT state for the computed delay period. After the message is sent out, the process again waits for input, and the cycle repeats. There is no internal buffering in a DELAY node for processes awaiting service. Once a message enters the node, it begins its wait period. However, it is buffered in the process while the process waits to send it out.

The particular semantics of this DELAY process models a single server in the sense that the delay time is only "charged" to a single message at a time. This is illustrated in the following example:

Suppose that some source of messages will try to send three messages to a DELAY node, which is initially ready to receive, with successive timestamps of 10, 20 and 80. Since the processes do not know how long they have been waiting to pass the message, all that can be asserted is that the arrivals to the DELAY node were at times at least as large as the three timestamps. Suppose that the delay values used for these three messages are 50, 5, and 30 respectively. The first message will leave the DELAY node at some time after time 60, because it arrived at time 10 or later, and had to wait out a delay of 50 time units. The second message will leave at time 65, or later. The reason for this is that in this type of DELAY node, a message does not enter and begin its waiting period until previous messages have left the node. The last message leaves the node at some time at least as great as 110, since the node was "free" to begin its delay (i.e. begin its processing) when it arrived at time 80, or later, and the delay used was 30 time units. This behavior is summarized in the following table:

Message	Arrival Time	Delay Time	Departure Time
msg 1	>=10	50	>= 60
msg 2	>=20	5	>= 65
msg 3	>=80	30	>= 110

This behavior is guaranteed by beginning the time delay for a message, M, at local time, DTIME, which is maintained as the maximum of the timestamp M had when it arrived, and that of the last message sent out. DTIME corresponds to local wall-clock time inside of a DELAY node, with the unusual property that the time displayed is always a lower bound of the real time. According to this clock, the arrivals take place at times 10,60, and 80.

A separate time accumulator, DSUMPTIME, keeps the sum of all the computed delays. Hence, DSUMPTIME is the total "busy" time of the node. The value DTIME - DSUMPTIME is a measure of the time lost because of starvation to the node. The ratio of DSUMPTIME to DTIME is roughly the utilization of the node.

If messages had fields representing their sizes, then the DELAY node could be constructed to return some function of the message size as its estimate of compute time. This provides a very easy way of simulating an algorithm whose time complexity is a function of problem size, e.g. O(size**2).

4.1.4 The QUEUE20 Process Type

The process that simulates a FIFO waiting queue in this implementation is named QUEUE20 because the procedure that implements the process is compiled with a hard limit of 20 as the maximum queue size that can be accomodated. The actual run time queue size maximum, called Q20MAX, is constrained to fall between one and 20, inclusive, and is solicited from the operator at process-creation.

The following quasi-CSP program is presented to demonstrate the translation of a program from the original CSP into the procedural description required by SIM.

```
QUEUE20 ::

     { LOCAL VARIABLE DECLARATION AND INITIALIZATION }

     Q20INPORT,Q20OUTPORT:PORTNAME;
     Q20INPTR,Q20OUTPTR:INTEGER;
     Q20BUFFER:ARRAY[0..19]OF INTEGER;

     { initialization of local variables --@ MPC = 1 }
     Q20INPTR:=1; Q20OUTPTR:=1;

     { PROCEDURAL DESCRIPTION                        }

     *  [
           { IF WE HAVE ROOM IN THE BUFFER }
           Q20INPTR<Q20OUTPTR+20;
           { THEN TRY TO RECEIVE A MESSAGE }
           Q20INPORT?Q20BUFFER[Q20INPTR MOD 20]  -->
           { WHEN RECEIVED, INCR INPUT PTR --@ MPC = 2 }
                  Q20INPTR:=Q20INPTR+1;
           { AT THE SAME TIME, }
        []
           { IF WE HAVE A MESSAGE TO SEND, }
           Q20INPTR>Q20OUTPTR;
           { TRY TO SEND IT OUT            }
           Q20OUTPORT!Q20BUFFER[Q20OUTPTR MOD 20] -->
           { WHEN SENT, INCR OUTPUT PTR --@ MPC = 3  }
                  Q20OUTPTR:=Q20OUTPTR+1                ]
```

Here it is assumed that the run-time queue size is always 20, and that naming a port for communication is equivalent to naming the process tied to the other end of the port. Q20INPORT is the name of the input port, and Q20OUTPORT is the name of the output port for this process. These are assumed to be bound to actual port numbers elsewhere.

The behavior of a QUEUE20 process is to initialize the local variables, and then any time it has a message to send out, it waits to send the message. At the same time, whenever there is room in the queue, the process waits to receive another message.

In the SIM implementation, the name of the process kind, QUEUE20, is bound at SIM compile time and is distributed throughout the program. The name appears in the list of allowable values for the scalar type PROCKIND, along with all other process names in the simulation. This same name, preceeded by "LP" (for logical process) is by convention the name of the Pascal procedure that emulates this process. "LPQUEUE20" also appears in the procedure RESUME, by means of which all processes are activated when in the simulation, they make a transition into the XQT state. See fig 3.2.

The process name also appears in the definition of the OWNDATA record type. This is where the definition of the local, or "own" variables, is bound to the type of the owning process kind, in this case, the QUEUE20.

As each instance of the process kind QUEUE20 is created, a fresh local address space, and activation record for the new QUEUE20 process are allocated in the form of previously unused elements in the OWN, and LPS arrays. The OWNDATA records, which comprise the base type of the OWN array, have their structure determined by the type of their owner.

An argument is provided in the calls to the CREATE pointers. Also at this label, if the queue is not full, the process waits for another input message.

Whenever a message is successfully sent out from this process, it is resumed at <u>MPC</u> = 3. Here it is certain that there is room for at least one more message, so the process waits for this input. Depending on the relative values of the pointers, there may be another message to send out, so the process may actually wait in parallel for both input and output.

4.1.5 The FORK2 Process Type

A process of kind FORK2 has a single input port, known as <u>F2INPORT</u>, and two output ports, known as <u>F2OUT1PORT</u>, and <u>F2OUT2PORT</u>. Initially, a FORK2 process waits for input. When this is received, the process decides which of its two output ports to send the message over, and begins waiting to do so. Once the message is sent out, the process again waits for input and the cycle repeats. Currently, the decision about which output port to try to send over is determined by performing a single Bernoulli trial, where the probability of a success, defined as deciding to send over <u>F2OUT1PORT</u>, is a parameter that is entered by the operator at process-create time.

A different decision policy would result in a process with the same connectivity topology, but different communication semantics. For example, the decision could be based on message traffic intensity, or perhaps an attribute of the particular message itself.

The chronological condition is satisfied because the inputs to the process are assumed to be chronological, and the process never holds more than one message, hence it has no opportunity to exchange two messages before they are sent out.

4.1.6 The MERGE2 Process Type

The MERGE2 process has two input ports, named M2IN1PORT, and M2IN2PORT, and a single output port named M2OUTPORT. The waiting rules of this type of process are pretty much fixed because of the requirement that the output messages be chronological, and the fact that there is minimal queueing inside of the process.

Initially, the process waits for input from either ports. As soon as any message is received over either port, it is buffered, and the process continues to wait for a message to come in over the other port. When this second message is received, the timestamps of the two messages are compared, and the process waits to send the message with the smaller timestamp over its output port. When the message is sent out, the process waits for input over the port which provided the message that was last sent. Once this message is received, the timestamps are again compared, and the cycle repeats. If two timestamps are found to be equal upon comparison, one message is arbitrarily selected to be sent first, and when it is sent, the process begins waiting for the port that provided this message.

This section has described six process varieties which have been modelled using SIM. Many others are possible, and some of these have been suggested. Each of these fills a role as a building block for constructing fairly general networks of processes. The processes as presented are useful for simulation of networks that create, delay, enqueue, and kill messages or jobs. The exact processing performed within a network depends on characteristic parts of the process' code.

4.2 Deadlocks in Distributed Simulation

A problem with distributed algorithms following the waiting rules inherent in the processes described above is the occurrence of deadlocks. Of course, deadlocks that arise in the modelled system will show up in the simulation; this is desirable. In fact, the purpose of a simulation might be to gain insights into the deadlock characteristics of a network. However, other deadlocks, not arising in the simulated system can and do show up strictly as a result of the waiting rules proposed in [CHA79].

4.3 Deadlock Resolution in Distributed Simulation

A solution proposed in this same paper was to introduce special "null" messages, whose sole purpose was to advance the line times of the ports, so that some process in a deadlocked system might be able to change the ports upon which it was waiting, while still guaranteeing the chronological condition for all of the ports, and hence break the deadlock. This algorithm was implemented in [SEE79] and found to work well in systems without feedback paths. In systems with feedback paths, the number of null messages grew so large as to flood the communication capacity of the network.

A separate solution was proposed in [CHA81] that computes a least upper bound on the line times. The algorithm consists of N phases, where a phase is comprised of an execution part followed by a communication part. N represents the total number of processes in the network. The reader is referred to the original article for a complete discussion of the problem of deadlocks in distributed simulation, with examples, and a description of the deadlock recovery algorithm together with a proof of its correctness.

As a testbed for general purpose distributed simulation, this algorithm has been implemented in the processes described above. During a simulation run, the processes compute and communicate until deadlock occurs, at which point, the entire system typically backs up and ceases to make progress. In the current implementation, the deadlock is detected by the central <u>CONTROLLER</u>, which initiates the recovery algorithm, and synchronizes the N phases. Initially, the line time used for each port is infinitely large. Then at each phase, every process either revises downward, or leaves constant its estimate of the earliest time (= smallest line time) at which it may try to send out a message over each of its output ports. At the k-th computation step, the process is able to do this based on the presense of the (k-1)st such estimate from all processes that send messages to it.

In a pure form, this would have been passed as a message to it along all of its input ports during the (k-1)st communication phase. This would be complicated in SIM, however, because the ports can only pass messages of a single type. Hence a pure implementation in SIM would require a network of communication lines for these time estimates that was parallel to the regular message ports. In the SIM implementation, this pure structure has been compromised to the extent of providing a dedicated word of the port's records to carry this information. This corresponds in the physical system to giving a little of the bandwidth of the port over to the deadlock recovery messages, as was needed. Of course, the ports would not be trying to carry regular messages at the same time, because if they were, the system would not be deadlocked.

A process executing its k-th step of this algorithm is resumed at MPC = 1010. There it reads the (k-1)st estimate on all of its input ports, and writes its own k-th estimate to its output ports. When the network consists of N processes, the algorithm computes its terminal values within N steps. Typically, the actual number of steps required is much lower, and seems tied to the length of the longest closed chain of processes.

After the N steps, the central CONTROLLER resumes the processes one last time in order to let them try to change the ports they are waiting for. According to [CHA81], there is always at least one process that can change its waiting pattern. At this point, if a process can change the lines that it is waiting for, it does so by means of PARWAIT calls.

After this k-step algorithm runs, the CONTROLLER attempts to make some of the ports fire, as if there had been no deadlock. If any firing is possible, at least one message is fired, and the simulation can continue. Otherwise, the deadlock cannot be eliminated, due to the fact that it is one which arose in the physical system, not one which appeared as a result of the waiting rules.

The results of SIM runs with this algorithm are encouraging. The algorithm works as expected to recover from deadlocks in both feedback, and feedforward networks. The interpretation of performance statistics for the algorithm requires only that the amounts of time for communication and processing be stated for a particular network. The relative cost would be greater in communication-limited, as opposed to compute-bound systems. As long as the time required for a process to communicate with all other N-1 processes is no worse than O(N), the overall performance is O(N**2), because it repeats this O(N) communication for each of the N steps of the algorithm. This result would be somewhat harder to establish for a totally distributed implementation, e.g. because of the time required to detect the deadlock initially, but should prove to be true.

As a last point on this subject, it is very easy to examine the effect of queue size on the frequency of deadlocks in such a distributed simulation. In an environment where execution of the recovery algorithm were very undesirable, one could see the advantage to be gained in avoided execution that could be obtained at the expense of extra memory buffering for the processes. The results from such a simulation for two particular networks appear in appendix 2.

5.0 SUGGESTIONS FOR FURTHER WORK

An obvious extension of the SIM program would be to provide the front-end language processing that would be required to implement a very useful subset of CSP. Such a translator would output entire SIM programs with the process-procedures embedded in them. This report has suggested some of the strategy that could be used in such a program.

In the current implementation, the topology of a network is constant throughout the execution of the simulation after it is entered during the network specification part of a run. This is not a general constraint imposed by SIM however. The connectivity of the network is contained in the global data of the system, and by changing this data during a run, the effective topology could be changed. In addition to changing the records about who talks to whom, new instances of the processes can be created, initialized, and executed, all on the fly. In this way, a network could change its topology to adapt to a particular processing problem.This would allow very general forking and joining in the processes of the network, as is required by many of the languages supporting concurrency. By spawning a number of subprocesses, a process could implement nondeterministic algorithms.

A very simple mapping would allow an instance of the SIM program and its processes to become the program that operated a true distributed system with a central controlling process, communicating e.g. over wires. The "MPC label" concept would map directly onto any of the typical machines with a vectored interrupt structure. With some modifications, the "oracle" functions provided by the CONTROLLER could be distributed and placed in the processes they served, possibly in the form of hardware bus arbitration.

6.0 SUMMARY

This report has described the SIM program, which has been written to assist one in the development of programs with distributed control in the form of a number of processes executing in parallel, and strong interactions in the form of messages passed between the processes. The program was motivated by the requirements of a language proposed by Hoare, that has proven useful in the expression of programs based on message passing.

Tha data, control structures, and special debug facilities of the program were detailed as an aid for anyone wishing to use it for future work.

A particular application, distributed simulation, was used as an example of the process structure and general nature of networks supported by SIM. Handling of the problem of deadlocks in this type of system was accomodated at modest expenditure of effort by introducing a dependence of the algorithm upon the central <u>CONTROLLER</u>.

APPENDIX 1

TEXT OF SIM PROGRAM

```
PROGRAM DSIM(INPUT*,OUTPUT);

(* IN THIS LISTING, 'TARDEP' MEANS TARGET DEPENDENT, AND IMPLIES
   THAT THE REFERENCED CODE OR DATA STRUCTURES ARE IMPACTED
   BY CHANGES IN THE TARGET NETWORK TO BE SIMULATED.          *)

CONST

    (* TARDEP * TOTAL NETWORK PORT COUNT *)
    PORTMAX = 100;

    (* TARDEP * TOTAL NETWORK PROCESS COUNT *)
    PROCMAX = 100;

 TYPE

    (* TARDEP * SCALAR PROCESS TYPE LIST*)
    PROCKIND = (SOURCE,SINK,FORK2,MERGE2,DELAY,QUEUE20);

    (* EACH LOGICAL PROCESS IS ALWAYS IN ONE OF FOUR STATES,
       ACTUALLY EXECUTING (XQT), WAITING FOR COMMUNICATION (BLK),
       ACTUALLY COMMUNICATING (CMN), OR TERMINATED (TRM).         *)
    STATETYPE = (XQT,BLK,TRM,CMN);

    (* SMALL INTEGERS PORTID AND PROCID TAKE ON VALUES THAT UNIQUELY
       IDENTIFY A PARTICULAR PORT OR PROCESS, RESPECTIVELY.           *)
    PORTID =  1..PORTMAX;
    PROCID =  1..PROCMAX;

    (* PORTDIRECTIONTYPES ARE USED AS PARAMETERS TO DEFINEPORT *)
    PORTDIRECTIONTYPE = (INN,OUT);

    (* TARDEP * SCALAR TARGET MESSAGE TYPE LIST*)
    MSGKIND = (UNDEFINED,JOB);

    (* TARDEP * TARGET MESSSAGE-RECORD TYPE.  A 'MESSAGE' CONSISTS OF
       AN MTIME FIELD FOR THE TIME COMPONENT OF THE MESSAGE, AND AN MKIND
       FIELD TO DENOTE THE TARGET DEPENDENT MSGKIND OF THE MESSAGE,
       E.G. 'SIGNAL' & 'ACK'.  THE MKIND VARIABLE ALSO DETERMINES THE
       REMAINING STRUCTURE OF THE MESSAGE.  EACH NEW ELEMENT IN THE
       SCALAR MSGKIND LIST MUST HAVE AN ASSOCIATED CASE MKIND
       DEFINITION IN THE FOLLOWING LINES.                               *)
    MESSAGE = RECORD
       MTIME:INTEGER;
       CASE MKIND : MSGKIND OF
          UNDEFINED : (UDATA:INTEGER);
          JOB       : (JOBNUMBER:INTEGER);
       END;

    (* EACH LOGICAL PROCESS HAS AN ASSOCIATED ACTIVITY RECORD OF TYPE
       ACTREC, THAT RETAINS THAT PART OF THE PROCESS' STATE VECTOR THAT
       IS MAINTAINED BY THE INTERPRETER.  READ 'A' FOR ACTIVITY IN
       INTERPRETING THE FIELDS.  THE TARGET PROCESSES INFORM THE
       INTERPRETER OF THEIR INTENDED NEXT STATE VIA THE VARIABLE
       ANEXTSTATE, WHICH ONLY HAS MEANINGFUL VALUES OF BLK OR TRM.
       ASTATE AND ANEXTSTATE THEN ARE THE CURRENT AND NEXT STATES OF
       THE BOUND PROCESS RESPECTIVELY.  THE ATIMELEFT FIELD HAS
       MEANING IN BOTH THE XQT AND CMN STATES, AND DENOTES THE AMOUNT
```

```
    OF TIME REQUIRED TO CONCLUDE THE CURRENT EXECUTION AND COM-
    MUNICATION RESPECTIVELY.  THE SUMS VARIABLES RETAIN THE AMOUNT
    OF TIME SPENT SO FAR IN EACH OF THE STATES XQT, BLK, AND CMN.
    IF THE CURRENT STATE OF A PROCESS IS TRM, THEN ATRMTIME HAS
    THE NETWORK CLOCK TIME THAT THE PROCESS BECAME TERMINATED.
    AMESSAGE IS A ONE-MESSAGE BUFFER USED FOR MESSAGES TRANSFERED
    TO AND FROM THE PORTS FOR COMMUNICATION.
    ATYPE AND AINSTANCE CONTAIN THE TARGET NETWORK PROCKIND AND
    INSTANCE OF THE BOUND PROCESS, E.G. 'SERVER' '7', OR 'MERGE' '2'.
    AMPC (FOR META PROGRAM COUNTER) IS THE MEANS OF TELLING THE
    INTERPRETER WHAT THE PROCESS POINT-OF-RETURN SHOULD BE FOLLOWING
    COMMUNICATION OVER A PARTICULAR PORT.  THIS IS PASSED AS A
    PARAMETER IN THE PARWAIT CALL THAT DECLARES THAT PORT TO BE
    ONE OF THE AWAITED PORTS FOR THE CALLING PROCESS DURING THE NEXT
    BLK STATE FOR THE PROCESS.
    ACCORDINGLY, WHEN THE INTERPRETER HAS CHOSEN A PARTICULAR PORT
    FOR FIRING, THE COMMUNICATING PROCESSES ARE INFORMED OF THEIR
    RETURN POINT THAT CORRESPONDS TO ACTION TO FOLLOW THIS PORT
    FIRING.                                                        *)
ACTREC = RECORD
    ATYPE:PROCKIND;
    AINSTANCE:1..PROCMAX;
    ASTATE,ANEXTSTATE:STATETYPE;
    ATIMELEFT,ASUMXQTTIME,ASUMCMNTIME,ASUMBLKTIME:INTEGER;
    ATRMTIME,AMPC,AXQTTIME:INTEGER;
    APORT:PORTID;
    AMESSAGE:MESSAGE;
    END;

(* EACH PORT HAS AN ASSOCIATED PORT RECORD OF TYPE PORTREC.
    READ 'P' FOR PORT IN INTERPRETING THE FIELD NAMES.
    A PORT HAS A FIXED SENDER AND RECEIVER WHICH ARE NAMED AS
    PSENDER AND PRECEIVER RESPECTIVELY.  FOR EACH OF SENDER AND
    RECEIVER, THE PORT MAINTAINS BOOLEAN VARIABLES INDICATING
    READY AND ELIGIBLE.  WHERE X IS ONE OF S FOR SENDER OR
    R FOR RECEIVER, PXWAITING <===> ( X IS READY TO COMMUNICATE ) ,
    AND PXELIGIBLE <===> ( X MARKED THIS PORT AS ELIGIBLE DURING
    ITS LAST EXECUTION PHASE BY MEANS OF A PARWAIT CALL ) .
    PORTS ARE NEVER FIRED UNLESS BOTH NAMED PROCESSES ARE
    BOTH ELIGIBLE AND WAITING.
    WHEN THE MESSAGE DOES FIRE, THE SENDER AND RECEIVER
    ARE INFORMED OF THEIR RESUME POINTS AS PSMPC AND PRMPC RESP.
    PSTIME AND PRTIME ARE THE NUMBER OF TIME UNITS REQUIRED TO
    SEND AND RECEIVE A MESSAGE ON THIS PORT.  COMMUNICATION TIME
    IS EXPLICITLY MODELLED WHEN THESE VALUES ARE SET NON-ZERO.
    A VIRTUAL FIELD, PCAPACITY, IS A MEASURE OF THE COMMUNICATION
    CAPACITY REQUIRED FOR THE 'LINE OF COMMUNICATION' MODELLED
    BY THIS PORT:
            PCAPACITY = (SIZE OF MESSAGE)/(TIME REQUIRED FOR TRANS)
    SIMILARLY, PDELAY, FOR PORT TRANSMISSION DELAY COULD BE:
            PDELAY    = PRTIME - PSTIME
    THAT IS, THE ADDITIONAL AMOUNT OF TIME THAT COMMUNICATION
    DETAINS THE RECEIVER COMPAIRED WITH THE SENDER.
    HENCE FAIRLY GENERAL COMMUNICATION CAN BE MODELLED EXPLICITLY.
    PWIJ IS USED IN THE DEADLOCK RECOVERY COMPUTATION OF W SUB I,J
    THIS IS BEING HANDLED FOR THESE EXPERIMENTS
    AS PART OF THE PORT RECORD INSTEAD OF THE LOCAL DATA OF THE LPS.
    AN IMPLEMENTATION MIGHT MULTIPLEX THE CHANNEL INTO A DATA PART,
    AND A PART USED TO COMMUNICATE THE INFORMATION NECESSARY TO
```

```
         BREAK DEADLOCK.                                                   *)

      PORTREC = RECORD
         PSENDER,PRECEIVER:PROCID;
         PSELIGIBLE,PSWAITING,PRWAITING,PRELIGIBLE:BOOLEAN;
         PSMPC,PRMPC,PSTIME,PRTIME,PMSGCOUNT:INTEGER;
         PMESSAGE:MESSAGE;
      PWIJ:INTEGER;
         END;

      (* TARDEP = PROCESS' OWN DATA RECORD TYPES
         EACH PROCESS HAS A DATA STRUCTURE FOR ITS OWN DATA, WHOSE
         STRUCTURE IS DETERMINED BY THE TARGET NETWORK PROCESS KIND
         E.G. A PROCESS OF KIND 'QUEUE' WOULD REQUIRE OWN DATA THAT
         RETAINED INFORMATION ABOUT JOBS IN THE QUEUE, AND HEAD AND
         TAIL POINTERS.  WHERE A BASIC PROCESS KIND HAS BEEN IN-
         STANTIATED SEVERAL TIMES, SEPARATE OWN DATA IS MAINTAINED
         FOR EACH ONE.
         EACH SUCH RECORD HAS AN OPROCKIND FIELD TO RECORD THE OWNER
         PROCESS KIND, AND A VARIABLE PART WHOSE STRUCTURE IS
         DETERMINED BY OPROCKIND.
         SEE THE DISCUSSION ABOUT THE VARIABLE, OWN, FOR EXAMPLES OF
         REFERENCING OWN DATA.                                              *)

      (* QUEUETYPE IS A TYPE DEFINED FOR USE BY THE QUEUE20 LP KIND *)
      QUEUETYPE = ARRAY[0..19] OF INTEGER;

      OWNDATA = RECORD
         CASE OPROCKIND : PROCKIND  OF
            SOURCE:(SOOUTPORT:PORTID;SOMU:REAL;SOTIME,SOMSGCOUNT:INTEGER;
               SOCON:REAL;SOSEQ:INTEGER);
            SINK:(SIINPORT:PORTID;SITIME,SIJOBCOUNT:INTEGER);
            MERGE2:(M2IN1PORT,M2IN2PORT,M2OUTPORT:PORTID;
               M2IN1TIME,M2IN2TIME,M2JOB1COUNT,
               M2JOB2COUNT:INTEGER;M2HAVE1,M2HAVE2:BOOLEAN;
               M2IN1MSG,M2IN2MSG:MESSAGE);
            FORK2:(F2INPORT,F2OUT1PORT,F2OUT2PORT:PORTID;
               F2TIME,F2OUT1COUNT,F2OUT2COUNT:INTEGER;F2RHO:REAL);
            QUEUE20:(Q20INPORT,Q20OUTPORT:PORTID;
               Q20BUFFER,Q20BUFFER2:QUEUETYPE;Q20TIME:INTEGER;
                  Q20INPTR,Q20OUTPTR,Q20JOBCOUNT,Q20QMAX:INTEGER);
            DELAY:(DINPORT,DOUTPORT:PORTID;DEMU,DCON:REAL;
               DTIME,DSUMPTIME,DJOBCOUNT:INTEGER);
         END;

       (* A TALLEYBYSTATE VARIABLE HAS 4 INTEGER FIELDS TO RETAIN A COUNT
          OF THE NUMBER OF PROCESSES IN EACH STATE.  SUCH A VARIABLE
          IS USED AS A RESULT PARAMETER OF THE COUNTBYSTATE ROUTINE WHICH
          COUNTS THE NUMBER OF PROCESSES IN EACH STATE AND PUTS THE TOTALS
          INTO THE APPROPRIATE FIELDS.  FOLLOWING THIS PROCEDURE CALL, THE
          COUNTS REMAIN VALID AS LONG AS NO PROCESS CHANGES STATE, AND
          NO ROUTINES CHANGE THE VALUES.  COUNTBYSTATE IS THE ONLY PLACE
          THESE VARIABLES ARE WRITTEN INTO.                                  *)
       TALLEYBYSTATE = RECORD
          XQTING,CMNING,BLKED,TRMED:INTEGER;
          END;

       (* A PORTPOINTER IS AN ARRAY WITH ONE ENTRY PER PORT, WHOSE
          ELEMENTS ARE POINTERS TO A PORT. ( SMALL INTEGER INDICIES INTO
```

```
       THE PORTS' DATA RECORD).  THIS IS A CONVENIENT MECHANISM TO
       DEFINE AN ORDERING ON THE PORTS SUCH AS THE USE BY THE SCHEDULE
       PROCEDURE TO DEFINE THE ORDER IN WHICH THE PORTS SHOULD FIRE. *)
    PORTPOINTER = ARRAY[1..PORTMAX] OF 0..PORTMAX;


VAR        (*  GLOBAL VARIABLES  *)


    (* PORTS IS A TABLE OF PORTRECORDS, ONE FOR EACH PORT IN THE
       TARGET NETWORK.  A PORT'S PORTID INDEXES THE TABLE.        *)
    PORTS : ARRAY[1..PORTMAX] OF PORTREC;

    (* LPS IS A TABLE OF ACTIVATION RECORDS, ONE FOR EACH LOGICAL
       PROCESS IN THE TARGET NETWORK.  A PROCESS' PROCID INDEXES
       INTO THE TABLE.                                            *)
    LPS   : ARRAY[1..PROCMAX] OF ACTREC;
    (* OWN IS A TABLE CONTAINING THE PROCESS' OWNED DATA RECORDS,
       ONE FOR EACH LOGICAL PROCESS. PROCESS I ACCESSES ITS DATA
       AS OWN[I].HEAD , OWN[I].TAIL AND OWN[I].Q[1]  FOR THE
       DEFINITIONS OF OWN DATA FOR THE PROCESS KIND 'QUEUE'
       DISCUSSED WITH THE OWNDATA TYPE DEFINITION.             *)
    OWN   : ARRAY[1..PROCMAX] OF OWNDATA;
    (* NETTIME IS THE GLOBAL NETWORK CLOCK.  A DISCRETE-VALUED
       CLOCK, IT ADVANCES ONE TIME UNIT EVERY TIME THE GLOBAL
       CLOCK 'TICKS', SEE THE PROCEDURE DEFINITION FOR TICK.

       NETSUMXQTTIME,NETSUMCMNTIME, AND NETSUMBLKTIME ACCUMULATE
       THE AMOUNT OF TIME THAT PROCESSES SPEND IN STATES
       EXECUTING, COMMUNICATING, AND BLOCKED RESPECTIVELY.

       NETMSGLIMIT AND NETTIMELIMIT ARE THE TIME AND MESSAGE BOUNDS
       THAT THE PROGRAM OPERATOR HAS SPECIFIED.  IF EITHER LIMIT
       IS EXCEEDED, THE SIMULATION IS STOPPEDAND A MESSAGE PRINTED.
       BEFORE THIS HAPPENS HOWEVER, THE OPERATOR GETS THE OPTION OF
       ENTERING NEW, HIGHER VALUES SO THAT THE SIMULATION MAY
       PROCEED.  NETMSGCOUNT CONTAINS THE TOTAL NUMBER OF MESSAGES
       THAT HAVE BEEN SENT SINCE THE BEGINNING OF THE SIMULATION.

       PORTJ IS AN INDEX VARIABLE THAT RANGES OVER THE DEFINED PORTS

       PROCI AND PROCK ARE INDEX VARIABLES THAT RANGE OVER THE PROCS

       NETDEADLOCK AND NETTERM ARE BOOLEAN VARIABLES WITH DEFINITIONS:
       ALL NONTERMINATED PROCESSES ARE DEADLOCKED, AND ALL PROCESSES
       ARE TERMINATED RESPECTIVELY.

       SEVEN DIFFERENT TRACE VARIABLES TURN ON SIMULATION RUN-TIME
       OUTPUT FOR THE PURPOSE OF DEBUGGING THE INTERPRETER OR THE
       TARGET LOGICAL PROCESSES, OR OBSERVING THE PROGRESS OF A
       SIMULATION.  IN GENERAL THESE VARIABLES PRODUCE MORE LISTING
       WHEN THEY HAVE LARGER VALUES.  ALL ARE INTEGER, AND A ZERO
       MEANS NO-TRACE.  THESE ARE:
         NAME          GOVERNS LISTING WITHIN THE SUBJECT AREA
       NETXQTTRACE   THE PROCESS STATE = EXECUTING
       NETCMNTRACE   THE PROCESS STATE = COMMUNICATING
       NETBLKTRACE   THE PROCESS STATE = BLOCKED
       NETTRMTRACE   THE PROCESS STATE = TERMINATED
       INTERTRACE    ACTION OF THE INTERPRETER, PROCEDURE CALLS, ETC.
```

```
      TARGETTRACE   ACTIVITIES OF THE TARGET NETWORK
      NETDEADTRACE  DEADLOACK DETECTION AND RECOVERY

      COUNT IS INITIALIZED BY COUNTBYSTATE AND PROVIDES A CONVENIENT
      WAY OF DETERMINING THE NUMBER OF PROCESSES THAT ARE IN EACH STATE
      AT ANY GIVEN TIME.  SEE THE TESTS FOR TERMINTION AND DEADLOCK
      DETECTION IN THE MAIN PROGRAM.

      TTY IS THE PROGRAM NAME OF THE OPERATOR'S I/O DEVICE, ASSUMED
      TO BE A TELETYPE-LIKE DEVICE.

      NETFAIR IS AN OWNED VARIABLE OF THE SCHEDULE PROCEDURE THAT
      IS USED TO ENSURE FAIRNESS AMONG THE WAITING PORTS, SO THAT
      NO MESSAGE WITH ELIGIBLE AND WAITING SENDER AND RECEIVER IS PASSED
      OVER INFINITELY OFTEN FOR FIRING.  SEE PROCEDURE SCHEDULE.

      DEADLOCKCOUNT IS A COUNT OF THE NUMBER OF DEADLOCKS DETECTED
      IN THE SEQUENTIAL SIMULATION.  NOTE THAT THIS INCLUDES ANY
      THAT ARISE IN THE TARGET SIMULATION, AND TYPICALLY MANY MORE
      THAT ARTIFICIALLY ARISE BECAUSE OF THE SEQUENTIAL SIMULATION
      OF THE WAITING RULES FOR MERGE PROCESSES, AND POSSIBLY OTHERS.

      HIGHPROC AND HIGHPORT ARE THE HIGHEST NUMBERED PROC AND PORT
      RESPECTIVELY, THAT ARE USED IN THE CURRENT SIMULATION.  THIS
      DEPENDS ON WHAT NETWORK THE OPERATOR HAS SPECIFIED.

                                                                  *)
   NETTIME,NETSUMXQTTIME,NETSUMCMNTIME,NETSUMBLKTIME:INTEGER;
   NETMSGLIMIT,NETTIMELIMIT,NETMSGCOUNT:INTEGER;
   PORTJ: 1..PORTMAX;
   PROCI,PROCK: 1..PROCMAX;
   NETDEADLOCK,NETTERM:BOOLEAN;
   NETCMNTRACE,NETXQTTRACE,NETBLKTRACE,INTERTRACE,NETTRMTRACE:INTEGER;
   TARGETTRACE:INTEGER;  COUNT:TALLEYBYSTATE;
   NETDEADTRACE:INTEGER;
   TTY : ATEXT;
   NETFAIR:PORTID;  (* PORT FAIRNESS ==> PROCESS FAIRNESS  *)
   DEADLOCKCOUNT:INTEGER;
   DEADLOCKLIMIT:INTEGER;
   HIGHPROC:PROCID;HIGHPORT:PORTID;
   BUFFERSIZE:INTEGER;   (* FOR BUFFERSIZE VS DEADLOCK EXPERIMENT *)

PROCEDURE CONTINUE;
(* PRINTS A MESSAGE AND SOLICITS DUMMY INPUT AS A DEBUG TOOL  *)
VAR DUMMY:CHAR;
BEGIN
   WRITELN(TTY,' CONTINUE CALLED, ENTER ANY CHAR ');
   BREAK;
   RESET(TTY);
   READ(TTY,DUMMY);
END;
FUNCTION MIN(ARG1,ARG2:INTEGER):INTEGER;
BEGIN
   IF ARG1<=ARG2 THEN MIN:=ARG1 ELSE MIN:=ARG2;
END;

FUNCTION MAX(ARG1,ARG2:INTEGER):INTEGER;
BEGIN
   IF ARG1>=ARG2 THEN MAX:=ARG1 ELSE MAX:=ARG2;
```

```
END;

PROCEDURE INCR(VAR ARG:INTEGER);
BEGIN
   ARG:=ARG+1;
END;

PROCEDURE DECR(VAR ARG:INTEGER);
BEGIN
   ARG:=ARG-1;
END;

PROCEDURE SHOWMSG(MSG:MESSAGE);
(* TARDEP * PROCEDURE TO WRITE THE PERTINANT INFORMATION FROM MSG
   TO THE TTY.  THE ALLOWABLE VALUES OF MSG.MSGKIND MUST BE ACCOUNTED
   FOR IN THE CASE STATEMENT SO THAT ALL ALLOWABLE MESSAGES CAN BE
   DUMPED TO TTY.                                                        *)
BEGIN
   WRITELN(TTY);
   WITH MSG DO BEGIN
      WRITE(TTY,' MTIME = ',MTIME:6,' HAS KIND =');
      CASE MKIND OF
         UNDEFINED:BEGIN
            WRITELN(TTY,' UNDEFINED, MESSAGE UDATA = ',UDATA);
            END;
         JOB :BEGIN
            WRITELN(TTY,' JOB, JOBNUMBER = ',JOBNUMBER);
            END;
         END;

      END;
END;

PROCEDURE COPYMSGTOFROM(VAR DEST:MESSAGE;FROM:MESSAGE);
(* TARDEP * COPIES THE FROM MESSAGE TO THE DEST MESSAGE.            *)
(* MUST BE ABLE TO COPY ALL THE NECESSARY PARTS OF ALL POSSIBLE
   MESSAGE VARIETIES, AS DETERMINED IN ANY PARTICULAR CALL BY THE
   VALUE OF FROM.MKIND   .                                          *)
BEGIN
   DEST.MTIME  := FROM.MTIME;
   DEST.MKIND  := FROM.MKIND;
   CASE FROM.MKIND OF
      UNDEFINED:DEST.UDATA:=FROM.UDATA;
      JOB      :DEST.JOBNUMBER:=FROM.JOBNUMBER;
      END;
   IF (INTERTRACE>0) THEN
      WRITELN(TTY,' MESSAGE COPIED');
   IF (INTERTRACE>50) OR (NETCMNTRACE>50) THEN CONTINUE;
END;

PROCEDURE SHOWPORT(ID:PORTID);
(* DUMPTTY STATE OF PORT                                   *)
BEGIN
   WITH PORTS[ID] DO BEGIN
      WRITELN(TTY);
      WRITE(TTY,' PORT ',ID:3,' PSENDER=',PSENDER:3,' PRECEIVER=');
      WRITELN(TTY,PRECEIVER:3,' PSELIGIBLE=',PSELIGIBLE);
      WRITE(TTY,' PRELIGIBLE=',PRELIGIBLE,' PSWAITING=',PSWAITING);
      WRITE(TTY,' PRWAITING=',PRWAITING,' PSMPC=',PSMPC:3,' PRMPC=');
```

```
        WRITELN(TTY,PRMPC:3);
        WRITELN(TTY,' PWIJ = ',PWIJ:6);
        WRITE(TTY,' PSTIME=',PSTIME:8,' PRTIME=',PRTIME:8);
        WRITELN(TTY,' PMSGCOUNT=',PMSGCOUNT:6,' PMESSAGE IS ');
        SHOWMSG(PMESSAGE);
    END;  (* WITH *)
END;

PROCEDURE PRSTATETYPE(ST:STATETYPE);
(*  PRINT PROCESS STATE GIVEN BY ST          FIELD WIDTH IS 5  *)
BEGIN
   CASE ST OF
        XQT:WRITE(TTY,' XQT ');
        BLK:WRITE(TTY,' BLK ');
        CMN:WRITE(TTY,' CMN ');
        TRM:WRITE(TTY,' TRM ');
        END;
END;

PROCEDURE PRMSGKIND(MK:MSGKIND);
(* TARDEP * WRITE THE MESSAGE KIND PRINTNAME AT THE CURRENT CURSOR
    ON TTY                                                                  *)
BEGIN
    CASE MK OF
        UNDEFINED:WRITE(TTY,' UNDEFINED ');
        JOB:WRITE(TTY,' JOB ');
    END;
END;


PROCEDURE PRPROCKIND(TY:PROCKIND);
(* TARDEP * PRINT LOGICAL PROCESS KIND GIVEN BY TY AT CURRENT CURSOR
    FIELD WIDTH IS 9.                                                       *)
BEGIN
    CASE TY OF
        SOURCE :WRITE(TTY,' SOURCE  ');
        SINK   :WRITE(TTY,' SINK    ');
        FORK2  :WRITE(TTY,' FORK2   ');
        MERGE2 :WRITE(TTY,' MERGE2  ');
        DELAY  :WRITE(TTY,' DELAY   ');
        QUEUE20:WRITE(TTY,' QUEUE20 ');
        END;
END;

PROCEDURE PRSIGNATURE(ID:PROCID);
(* PRINT PROCESS KIND, INSTANCE, AND PROCESS ID FOR THE PROCESS
    NUMBER PASSED AS ID, AT THE CURRENT CURSOR PUDITION.  FIELD WIDTH
    IS  31                                                                  *)
BEGIN
    PRPROCKIND(LPS[ID].ATYPE);
    WRITE(TTY,' INSTANCE = ',LPS[ID].AINSTANCE:4,' UNIQUE ID = ',ID:4);
END;

PROCEDURE SHOWPROCESS(ID:PROCID);
(* DUMPTTY STATE OF THE PROCESS NAMED ID.                        *)
BEGIN
    WRITELN(TTY);
    WRITE(TTY,' SHOWPROCESS');
    PRSIGNATURE(ID);
```

```
   WITH LPS[ID] DO BEGIN
      WRITELN(TTY);
      WRITE(TTY,' ASTATE =');
      PRSTATETYPE(ASTATE);
      WRITE(TTY,' ANEXTSTATE =');
      PRSTATETYPE(ANEXTSTATE);
      WRITELN(TTY);
      WRITE(TTY,' ATIMELEFT = ',ATIMELEFT:4,' ASUMXQTTIME = ');
      WRITELN(TTY,ASUMXQTTIME:6,' ASUMCHNTIME = ',ASUMCHNTIME:6);
      WRITE(TTY,' ASUMBLKTIME = ',ASUMBLKTIME:6);
      WRITE(TTY,' ATRMTIME = ',ATRMTIME:8,' AMPC = ',AMPC:4);
      WRITELN(TTY,' AXQTTIME = ',AXQTTIME:6);
      WRITELN(TTY,' ATIMELEFT = ',ATIMELEFT:5);
      WRITE(TTY,' APORT = ',APORT:4,' MESSAGE BUFFER ');
      WRITELN(TTY,'CONTAINS THE MESSAGE');
      SHOWMSG(AMESSAGE);
      END;
END;

PROCEDURE SHOWOWN(PROCS:PROCID);
(* TARDEP * PRINT OWNED DATA FOR PROCESS GIVEN BY PROCS      *)
VAR I:INTEGER;
BEGIN
   WRITELN(TTY);
   WRITE(TTY,' PROCESS ',PROCS:4,' HAS OWNED DATA FOR OPROCKIND=');
   PRPROCKIND(OWN[PROCS],OPROCKIND);
   WRITELN(TTY);
   WITH OWN[PROCS] DO
      CASE OPROCKIND OF
         SOURCE:BEGIN
            WRITE(TTY,' SOOUTPORT = ',SOOUTPORT:3);
            WRITE(TTY,' SOCON =',SOCON);
            WRITELN(TTY,' SOMU = ',SOMU);
            WRITE(TTY,' SOTIME = ');
            WRITE(TTY,' ',SOTIME:6,' SOMSGCOUNT = ',SOMSGCOUNT:6);
            WRITELN(TTY,' SOSEQ = ',SOSEQ:6);
            END;
         SINK:BEGIN
            WRITE(TTY,' SIINPORT = ',SIINPORT:3,' LOCAL TIME = ');
            WRITELN(TTY,SITIME:6,' SIJOBCOUNT = ',SIJOBCOUNT:6);
            END;
         QUEUE20:BEGIN
            WRITE(TTY,' Q20INPORT = ',Q20INPORT:3,' Q20OUTPORT = ');
            WRITELN(TTY,Q20OUTPORT:3,' Q20TIME = ',Q20TIME:6);
            WRITE(TTY,' Q20INPTR = ',Q20INPTR:7,' Q20OUTPTR = ');
            WRITELN(TTY,Q20OUTPTR:7,' Q20JOBCOUNT = ',Q20JOBCOUNT:5);
            Q20QMAX:=MAX(0,Q20QMAX);
            Q20QMAX:=MIN(19,Q20QMAX);
            WRITELN(TTY,' Q20QMAX = ',Q20QMAX:2);
            IF (TARGETTRACE>10) THEN  BEGIN
               FOR I:=0 TO Q20QMAX DO BEGIN
                  WRITELN(TTY,' Q20BUFFER[',I:2,'] CONTAINS MESSAGE :');
                  WRITE(TTY,' MESSAGE TIME = ',Q20BUFFER[I]:7);
                  WRITELN(TTY,' JOB SEQUENCE NUMBER = ',
                      Q20BUFFER2[I]:4);
                  END;
               WRITE(TTY,' Q20BUFFER CONTAINS ',Q20INPTR-Q20OUTPTR:2);
               WRITELN(TTY,' JOBS ');
               END;
```

```
           END;
        MERGE2:BEGIN
           WRITE(TTY,' M2IN1PORT = ',M2IN1PORT:3,' M2IN2PORT = ');
           WRITELN(TTY,M2IN2PORT:3,' M2OUTPORT = ',M2OUTPORT:3);
           WRITE(TTY,' M2IN1TIME = ',M2IN1TIME:6,' M2IN2TIME = ');
           WRITELN(TTY,M2IN2TIME:6);
           WRITE(TTY,' M2HAVE1 = ',M2HAVE1);
           WRITELN(TTY,' M2HAVE2 = ',M2HAVE2);
           WRITE(TTY,' M2JOB1COUNT = ',M2JOB1COUNT:6);
           WRITELN(TTY,' M2JOB2COUNT = ',M2JOB2COUNT:6);
           IF (TARGETTRACE>20) THEN BEGIN
              WRITELN(TTY,' M2IN1MSG CONTAINS THE MESSAGE :  ');
              SHOWMSG(M2IN1MSG);
              WRITELN(TTY,' M2IN2MSG CONTAINS THE MESSAGE :  ');
              SHOWMSG(M2IN2MSG);
              END;
           END;
        FORK2:BEGIN
           WRITE(TTY,' F2INPORT = ',F2INPORT:3,' F2OUT1PORT = ');
           WRITELN(TTY,F2OUT1PORT:3,' F2OUT2PORT = ',F2OUT2PORT:3);
           WRITE(TTY,' F2TIME = ',F2TIME:6,' F2OUT1COUNT = ');
           WRITELN(TTY,F2OUT1COUNT:6,' F2OUT2COUNT = ',F2OUT2COUNT:6);
           WRITELN(TTY,' F2RHO = ',F2RHO);
           END;
        DELAY:BEGIN
           WRITE(TTY,' DINPORT = ',DINPORT:3,' DOUTPORT = ');
           WRITELN(TTY,DOUTPORT:3,' DTIME = ',DTIME:6);
           WRITE(TTY,' DJOBCOUNT = ',DJOBCOUNT:6,' DEMU =',DEMU);
           WRITELN(TTY,' DCON = ',DCON);
           WRITELN(TTY,' DSUMPTIME = ',DSUMPTIME:6);
           END;
      END;  (* CASE  *)
   WRITELN(TTY);
END;

PROCEDURE SHOWNETWORK;
(* SHOWS THE PROCESS IDENTITIES, TYPES AND INSTANCES, AND CONNECTIVITY
   OF PROCESSES AND PORTS                                              *)
VAR PROCI:PROCID;PORTJ:PORTID;
BEGIN
   WRITELN(TTY);
   WRITELN(TTY,' NETWORK DEFINITION FOLLOWS, HERE ARE THE PROCESSES ');
   FOR PROCI:=1 TO HIGHPROC DO BEGIN
      WRITE(TTY,'  ');
      PRSIGNATURE(PROCI);
      WRITELN(TTY);
      END;
   WRITELN(TTY);
   WRITELN(TTY,' HERE ARE THE PORTS ');
   WRITE(TTY,'  SENDER   RECEIVER   PORTID   MSGKIND ');
   WRITELN(TTY);
   FOR PORTJ:=1 TO HIGHPORT DO WITH PORTS[PORTJ] DO BEGIN
      WRITE(TTY,'     ',PSENDER:3,'       ',PRECEIVER:3);
      WRITE(TTY,'          ',PORTJ:3,'      ');
      PRMSGKIND(PMESSAGE,MKIND);
      WRITELN(TTY);
      END;
   WRITELN(TTY);
END;
```

```

PROCEDURE PARWAIT(WHICHPORT:PORTID;REQUESTER:PROCID;MPC:INTEGER);
(* REQUESTING PROCESS INVOKES PROCEDURE PARWAIT TO INITIATE
   PARALLEL WAITING FOR A MESSAGE OVER PORT # WHICHPORT.
   SHOULD THIS BE THE PORT OVER WHICH THIS PROCESS NEXT
   COMMUNICATES, EXECUTION SHOULD RESUME AT THE CONTROL
   POINT MPC ( META PROGRAM COUNTER ).  THIS PROC WILL
   CHECK THAT THE REQUESTING PROCESS IS NAMED AS A PARTY
   TO COMMUNICATION OVER THE NAMED PORT, AND THAT THE TYPE OF
   MESSAGE SENT TO THIS PORT AGREES WITH THE TYPE NAMED IN THIS
   PORT'S PORTREC.  AN ERROR MESSAGE IS PRINTED UPON CALL
   IF THESE REQUIREMENTS ARE NOT SATISFIED.                   *)

BEGIN
   IF (INTERTRACE+NETBLKTRACE+NETCMNTRACE+TARGETTRACE) > 0 THEN
      WRITE(TTY,' PARWAIT CALLED');
   IF (INTERTRACE>10) OR (NETBLKTRACE>10) OR (TARGETTRACE>10) THEN BEGIN
      WRITE(TTY,', PROCESS ',REQUESTER:3,' WILL AWAIT PORT ',WHICHPORT:3);
      IF (PORTS[WHICHPORT].PSENDER = REQUESTER) THEN BEGIN
         WRITE(TTY,' TO SEND A MESSAGE WITH MTIME = ');
         WRITELN(TTY,LPS[REQUESTER].AMESSAGE.MTIME);
         END;
      WRITE(TTY,' AT NETTIME = ',NETTIME+LPS[REQUESTER].AXQTTIME:6);
      WRITELN(TTY,' WITH RESUME POINT ',MPC:4);
      END;
   IF (PORTS[WHICHPORT].PSENDER=REQUESTER) AND
      (PORTS[WHICHPORT].PMESSAGE.MKIND<>LPS[REQUESTER].AMESSAGE.MKIND)
         THEN BEGIN
            WRITE(TTY,' SERIOUS ERROR ***  PROCESS ',REQUESTER:3);
            WRITELN(TTY,' HAS ALTERED THE MESSAGE TYPE OF PORT ');
            WRITE(TTY,' NUMBER ',WHICHPORT:3,' EFFECTIVELY CHANGING ');
            WRITELN(TTY,'ITS TYPE ** ERROR DETECTED IN PARWAIT');
            SHOWPROCESS(REQUESTER);
            SHOWPORT(WHICHPORT);
            WRITE(TTY,' THIS VIOLATES AN INVARIANT OF THE PORT');
            WRITELN(TTY,', FIXED MESSAGE TYPE');
            END;
   IF PORTS[WHICHPORT].PSENDER = REQUESTER THEN BEGIN   (* SEND *)
      COPYMSGTOFROM(PORTS[WHICHPORT].PMESSAGE,LPS[REQUESTER].AMESSAGE);
      PORTS[WHICHPORT].PSELIGIBLE := TRUE;
      PORTS[WHICHPORT].PSWAITING := FALSE;
      PORTS[WHICHPORT].PSMPC := MPC;
      LPS[REQUESTER].ANEXTSTATE := BLK;
      END  ELSE
         IF PORTS[WHICHPORT].PRECEIVER = REQUESTER THEN BEGIN
            (* REQUEST TO RECEIVE OVER THIS LINE *)
            PORTS[WHICHPORT].PRELIGIBLE := TRUE;
            PORTS[WHICHPORT].PRMPC := MPC;
            LPS[REQUESTER].ANEXTSTATE := BLK;
            END ELSE BEGIN
               WRITE(' THIS ILLEGAL REQUEST IGNORED');
               WRITE(TTY,', REQUESTER ',REQUESTER:3,' IS NOT ');
               WRITELN(TTY,' CONNECTED');
               WRITE(TTY,' TO THE PORT=',WHICHPORT:3,' NAMED IN');
               WRITELN(TTY,' PARWAIT CALL');
               END;
   IF (PORTS[WHICHPORT].PSENDER=REQUESTER) AND (NETCMNTRACE>10) THEN BEGIN
      WRITELN(TTY,' THE MESSAGE TO BE SENT IS');
      SHOWMSG(LPS[REQUESTER].AMESSAGE);
```

procedure that creates the processes, which should be unique to each call. The value of this "uniqueid" becomes the process-identifier of the resulting process created. This is the value that is used as the index into the OWN and LPS arrays for the process. If all the processes are well behaved, and never access the OWN and LPS arrays with arguments other than their own unique identifier, then some level of protection is maintained for the process' local data. Since processes are resumed with this same argument as a calling parameter, "ID", some enforcement of protection would be afforded by disallowing any reference to OWN or LPS with arguments other than this ID, and also flagging as an error any attempt to alter the value of this variable within a process. It would also be necessary to keep the OPROCKIND field of the OWNDATA entry for the process from changing during execution since this would alter the structure of the local data, possibly providing access to the data of another process, so any code that attempted to change this value should be disallowed.

In the SIM QUEUE20 type, initialization occurs at MPC = 1. Here it can be asserted that there is room in the queue; in fact, it is empty. Accordingly, the process initially waits to receive its first message. Whenever input is received, the process resumes at MPC = 2. At this label, it is known that there is at least one message to send, so the process can wait to send it out even without checking the

```
      END;
   IF (INTERTRACE+NETBLKTRACE+NETCMNTRACE+TARGETTRACE >0) THEN BEGIN
      WRITELN(TTY,' PARWAIT RETURNED ');
      END;
END;


PROCEDURE DUMPTTY;
(* DISPLAY THE WORLD                                      *)
VAR PROCI:PROCID;PORTJ:PORTID;
BEGIN
   WRITELN(TTY);
   WRITELN(TTY,' ** PROCEDURE DUMPTTY CALLED **  HERE ARE THE PORTS');
   FOR PORTJ:=1 TO HIGHPORT DO SHOWPORT(PORTJ);
   WRITELN(TTY,' HERE ARE THE PROCESS'' RECORDS');
   FOR PROCI := 1 TO HIGHPROC DO BEGIN
      SHOWPROCESS(PROCI);
      SHOWOWN(PROCI);
      END;
   WRITELN(TTY);
   WRITE(TTY,' NETTIME=',NETTIME:8,' NETSUMXQTTIME=');
   WRITE(TTY,NETSUMXQTTIME:8,' NETSUMCMNTIME=',NETSUMCMNTIME:8);
   WRITELN(TTY,' NETSUMBLKTIME=',NETSUMBLKTIME:8);
   WRITE(TTY,' NETMSGLIMIT=',NETMSGLIMIT:6,' NETTIMELIMIT=');
   WRITE(TTY,NETTIMELIMIT:8,' NETMSGCOUNT=',NETMSGCOUNT:6);
   WRITELN(TTY,' PORTJ=',PORTJ:4,' PROCI=',PROCI:4);
   WRITE(TTY,' NETDEADLOCK= ',NETDEADLOCK,' NETTERM= ',NETTERM);
   WRITELN(TTY,' NETCMNTRACE=',NETCMNTRACE:3);
   WRITE(TTY, ' NETXQTTRACE=',NETXQTTRACE:3,' NETBLKTRACE=');
   WRITELN(TTY,' INTERTRACE=',INTERTRACE:3);
   WRITELN(TTY,' DEADLOCKCOUNT = ',DEADLOCKCOUNT:5);
   WRITELN(TTY,' END DUMPTTY ');
END;

PROCEDURE TRACELP(ID:PROCID);
(* PRINT INFORMATION ABOUT THE STATE OF THE LOGICAL PROCESS NAMED
   ID.  MORE INFORMATION IS PRINTED THE LARGER THE CALLING VALUES
   OF THE TRACE VARIABLES.  NOTHING IS PRINTED AT ALL IF ALL TRACE
   VARS ARE ZERO.  CALLERS SHOULD PRINT PERTINANT DETAILS ABOUT
   THE LOCATION FROM WHICH THIS ROUTINE IS CALLED, E.G. WHEN
   CALLED AT THE BEGINNING OF  THE PROCEDURAL DESCRIPTION OF AN LP,
   THE CODE MIGHT BE:
          WRITE(TTY,' ENTER TARGET PROCESS');
          TRACELP(ID);
          WRITELN(TTY);
          END;
                                                                   *)
BEGIN
   IF (INTERTRACE>0) OR (TARGETTRACE>0) OR (NETXQTTRACE>0) THEN BEGIN
      PRSIGNATURE(ID);
      WRITELN(TTY);
      END;
   IF (INTERTRACE>9 ) THEN SHOWPROCESS(ID);
   IF (TARGETTRACE>9) THEN  SHOWOWN(ID);
END;

PROCEDURE DEFINEPORT(DIRECTION:PORTDIRECTIONTYPE;OWNER:PROCID;
      VAR PORTNUM:PORTID);
(* THIS ROUTINE WILL SOLICIT INFORMATION FROM THE USER ABOUT
```

```
    A PORT WHOSE DIRECTION, ONE OF INN OR OUT, IS PASSED IN AS
    DIRECTION, AND WHOSE OWNER PROCESS IS NAMED AS OWNER.  THE
    PORTNUMBER WILL BE RETURNED AS PORTNUM, AND THE PORTS ARRAY
    WILL BE UPDATED WITH THE USER-SUPPLIED DATA.  THIS INCLUDES
    THE PORTID, MESSAGE KIND, AND THE SEND OR RECEIVE TIME AS
    DETERMINED BY WHETHER THIS IS AN INN, OR AN OUT PORT.      *)

VAR INDATA,I:INTEGER;KIND:MSGKIND;
BEGIN
    WRITELN(TTY,' ENTER PORT ID NUMBER');
    BREAK;
    RESET(TTY);
    READ(TTY,PORTNUM);
    (*   DON'T GET THS FROM OPERATOR ...
    WRITE(TTY,' ENTER THE INTEGER CODE FOR THE TYPE OF MESSAGE ');
    WRITELN(TTY,'SENT OVER THIS PORT');
    I:=0;
    (* TARDEP * NEXT LINE CONTAINS 1ST AND LAST MSGKIND SCALARS  ENDCOMMENT
    FOR KIND:=UNDEFINED TO JOB DO BEGIN
        WRITE(TTY,'      ',I:2,' = ');
        PRMSGKIND(KIND);
        INCR(I);
        END;
    WRITELN(TTY);
    BREAK;
    RESET(TTY);
    READ(TTY,INDATA);
                                                                      *)
    INDATA:=1;
    CASE INDATA OF
        (* ALL MESSAGE KINDS MUST BE ACCOUNTED FOR IN CASE RANGE *)
        0:KIND:=UNDEFINED;
        1:KIND:=JOB;
        END;
    (* WRITE(TTY,' ENTER TIME UNITS FOR ');
    CASE DIRECTION OF
        INN:WRITELN(TTY,'INPUT FROM THIS PORT ');
        OUT:WRITELN(TTY,'OUTPUT TO THIS PORT ');
        END;
    BREAK;
    RESET(TTY);
    READ(TTY,INDATA);    *)
    INDATA:=1;
    WITH PORTS[PORTNUM] DO BEGIN
        CASE DIRECTION OF
            INN:BEGIN
                PRECEIVER:=OWNER;
                PRTIME:=INDATA;
                END;
            OUT:BEGIN
                PSENDER:=OWNER;
                PSTIME:=INDATA;
                END;
            END;
        PMESSAGE.MKIND:=KIND;
        END;
    IF (INTERTRACE>30) OR (TARGETTRACE>30) THEN BEGIN
        WRITE(TTY,' PORT ',PORTNUM:3,' IS INCIDENT ON');
        PRSIGNATURE(OWNER);
```

```
      WRITELN(TTY,' HERE IS THE PORT');
      SHOWPORT(PORTNUM);
      END;
END;

FUNCTION AWAITS(PROC:PROCID;PORT:PORTID):BOOLEAN;
(* RETURNS TRUE IFF PROC IS WAITING FOR COMMUNICATION OVER PORT *)
BEGIN
   WITH PORTS[PORT] DO
      AWAITS:=(((PRECEIVER=PROC) AND PRWAITING AND PRELIGIBLE) OR
               ((PSENDER=PROC)   AND PSWAITING AND PSELIGIBLE));
END;


(*****************************************************************
 *                                                               *
 *    DEFINITION OF TARGET PROCESSES VIA PASCAL PROCEDURES       *
 *                                                               *
 *****************************************************************)

PROCEDURE LPSOURCE(ID:PROCID);
(* THE TYPE OF THIS LOGICAL PROCESS IS OBTAINED BY REMOVING THE
   'LP' FROM ITS NAME.  THIS DETERMINES THE OWNDATA ENTRY FOR
   PROCESSES OF THE TYPE DESCRIBED BY THE FOLLOWING CODE.     *)

VAR DELAY:INTEGER;

PROCEDURE CREATEMESSAGE(VAR MSG:MESSAGE;TIME:INTEGER;KIND:MSGKIND;
                 DATA:INTEGER);
(* INITIALIZE THE MESSAGE WITH THE SUPPLIED TIME, KIND, AND SINGLE
   WORD OF DATA.                                                  *)
BEGIN
   WITH MSG DO BEGIN
      MTIME:=TIME;
      MKIND:=KIND;
      CASE MKIND OF    (* TARDEP * DEPENDING ON KIND OF MSG *)
         (* MUST ACCOUNT FOR EVERY POSSIBLE VALUE OF KIND EXP LATER *)
         JOB:JOBNUMBER:=DATA;
         UNDEFINED:UDATA:=DATA;
         END;
      END;
END;


BEGIN
   IF(TARGETTRACE>0) OR (INTERTRACE>0) THEN BEGIN
      WRITE(TTY,' (RE)-ENTERING ');
      PRSIGNATURE(ID);WRITELN(TTY);END;
   WITH LPS[ID] DO
      WITH OWN[ID] DO BEGIN
         AXQTTIME:=1;
         CASE AMPC OF
            0:BEGIN    (* CREATE THIS LP SOURCE INSTANCE  *)
                  WRITE(TTY,' CREATE');
                  PRSIGNATURE(ID);
                  WRITELN(TTY);
                  WRITE(TTY,' FOR THE OUTPUT PORT,');
                  DEFINEPORT(OUT,ID,SOOUTPORT);
                  WRITE(TTY,' ENTER REAL EXPONENTIAL DELAY ');
```

```
                WRITELN(TTY,' PARAMETER SOMU ');
                BREAK;
                RESET(TTY);
                READ(TTY,SOMU);
                WRITE(TTY,' ENTER REAL CONSTANT DELAY ');
                WRITELN(TTY,'PARAMETER SOCON');
                BREAK;
                RESET(TTY);
                READ(TTY,SOCON);
                END;
            1:BEGIN     (* FIRST CALL TO THE PROCESS NUMBER ID *)
               SOMSGCOUNT:=0;
               DELAY:=TRUNC(-SOMU*LN(RANDOM(0))) + TRUNC(SOCON);
               IF (SOCON=0.0) THEN INCR(DELAY);
               AXQTTIME:=DELAY;
               SOTIME:=DELAY;
               SOSEQ:=1;
               CREATEMESSAGE(AMESSAGE,SOTIME,JOB,SOSEQ);
               PARWAIT(SOOUTPORT,ID,2);
               END;
            2:BEGIN    (* JUST SENT OVER SOOUTPORT *)
               DELAY:=TRUNC(-SOMU*LN(RANDOM(0))) + TRUNC(SOCON);
               IF (SOCON=0.0) THEN INCR(DELAY);
               AXQTTIME:=DELAY;
               SOTIME:=SOTIME+DELAY;
               INCR(SOMSGCOUNT);
               INCR(SOSEQ);
               CREATEMESSAGE(AMESSAGE,SOTIME,JOB,SOSEQ);
               PARWAIT(SOOUTPORT,ID,2);
               END;
            1000:BEGIN  (* LAST CALL FOR ANY STATISTICS *)
               WRITE(TTY,' REPORT FROM');
               PRSIGNATURE(ID);
               WRITELN(TTY);
               SHOWOWN(ID);
               END;
            1010:BEGIN  (* W-SUB-IJ COMPUTATION *)
               WITH PORTS[SOOUTPORT] DO BEGIN
                  (* A SOURCE ALWAYS WAITS TO OUTPUT *)
                  PWIJ:=SOTIME;
                  END;
               END;
            1020:;      (* NO ACTION REQUIRED AFTER WIJ FOR SOURCE *)
            OTHERS:BEGIN
                  PRSIGNATURE(ID);
                  WRITELN(TTY,' CALLED WITH BAD AMPC = ',AMPC:4);
                  SHOWPROCESS(ID);
                  SHOWOWN(ID);
                  IF (INTERTRACE>20) OR (TARGETTRACE>20) THEN DUMPTTY;
                  END;
            END;        (* CASE *)
         END;
   IF(INTERTRACE>0) OR (TARGETTRACE>0) THEN BEGIN
      WRITE(TTY,' LEAVING  ');
      TRACELP(ID);
      END;
END;  (* LPSOURCE PROCEDURAL DESCRIPTION *)

PROCEDURE LPSINK(ID:PROCID);
```

```
(* THE TYPE OF THIS LOGICAL PROCESS IS OBTAINED BY REMOVING THE
   'LP' FROM ITS NAME.  THIS DETERMINES THE OWNDATA ENTRY FOR
   PROCESSES OF THE TYPE DESCRIBED BY THE FOLLOWING CODE.     *)

VAR DELAY:INTEGER;

PROCEDURE CONSUMEMESSAGE(VAR MSG:MESSAGE);
BEGIN
   (* TARDEP * DO AS YOU WILL WITH THIS MESSAGE BEFORE IT DIES *)
   MSG.MKIND:=UNDEFINED;
   MSG.MTIME:=0;
END;


BEGIN
    IF(TARGETTRACE>0) OR (INTERTRACE>0) THEN BEGIN
       WRITE(TTY,' (RE)-ENTERING ');
       PRSIGNATURE(ID);WRITELN(TTY);END;
    WITH LPS[ID] DO
       WITH OWN[ID] DO BEGIN
          AXQTTIME:=1;
          CASE AMPC OF
             0:BEGIN    (* CREATE THIS INSTANCE OF A SINK LP     *)
                  WRITE(TTY,' CREATE');
                  PRSIGNATURE(ID);
                  WRITELN(TTY);
                  WRITE(TTY,' FOR THE INPUT PORT,');
                  DEFINEPORT(INN,ID,SIINPORT);
                  END;
             1:BEGIN    (* FIRST CALL TO THIS SINK PROCESS *)
                  SITIME:=0;
                  SIJOBCOUNT:=0;
                  PARWAIT(SIINPORT,ID,2);
                  END;
             2:BEGIN    (* JUST RECEIVED A JOB, DO SOMETHING & DISCARD *)
                  SITIME:=AMESSAGE.MTIME;
                  INCR(SIJOBCOUNT);
                  CONSUMEMESSAGE(AMESSAGE);
                  PARWAIT(SIINPORT,ID,2);
                  END;
             1000:BEGIN  (* LAST CALL, PRINT LOCAL REPORT   *)
                  WRITE(TTY,' REPORT FROM');
                  PRSIGNATURE(ID);
                  WRITELN(TTY);
                  SHOWOWN(ID);
                  END;
             1010:;  (* SINK COMPUTES NO W SUB IJ FOR ANY PORTS   *)
             1020:;  (* AND CANNOT CHANGE THE AWAITED PORTS *)
             OTHERS:BEGIN
                  PRSIGNATURE(ID);
                  WRITELN(TTY,' CALLED WITH BAD AMPC = ',AMPC:4);
                  SHOWPROCESS(ID);
                  SHOWOWN(ID);
                  IF (INTERTRACE>20) OR (TARGETTRACE>20) THEN DUMPTTY;
                  END;
             END;        (* CASE *)
          END;
     IF (INTERTRACE>0) OR (TARGETTRACE>0) THEN BEGIN
        WRITE(TTY,' LEAVING ');
```

```
            TRACELP(ID);
            END;
    END;  (* LPSINK PROCEDURAL DESCRIPTION *)



    PROCEDURE LPDELAY(ID:PROCID);
    (* THE TYPE OF THIS LOGICAL PROCESS IS OBTAINED BY REMOVING THE
       'LP' FROM ITS NAME.  THIS DETERMINES THE OWNDATA ENTRY FOR
       PROCESSES OF THE TYPE DESCRIBED BY THE FOLLOWING CODE.     *)

    VAR DELAY:INTEGER;

    BEGIN
        IF(TARGETTRACE>0) OR (INTERTRACE>0) THEN BEGIN
            WRITE(TTY,' (RE)-ENTERING ');
            PRSIGNATURE(ID);WRITELN(TTY);END;
        WITH LPS[ID] DO
            WITH OWN[ID] DO BEGIN
                AXQTTIME:=1;
                CASE AMPC OF
                    0:BEGIN  (* CREATE THIS LP SINK INSTANCE *)
                          WRITE(TTY,' CREATE');
                          PRSIGNATURE(ID);
                          WRITELN(TTY);
                          WRITE(TTY,' FOR INPUT PORT,');
                          DEFINEPORT(INN,ID,DINPORT);
                          WRITE(TTY,' FOR OUTPUT PORT,');
                          DEFINEPORT(OUT,ID,DOUTPORT);
                          WRITELN(TTY,' ENTER CONSTANT DELAY REAL PARAMETER DCON');
                          BREAK;
                          RESET(TTY);
                          READ(TTY,DCON);
                          WRITELN(TTY,' ENTER EXPONENTIAL DELAY REAL PARAMETER DEMU');
                          BREAK;
                          RESET(TTY);
                          READ(TTY,DEMU);
                          END;
                    1:BEGIN   (* FIRST CALL TO THIS PROCESS *)
                          DSUMPTIME:=0;
                          DTIME:=0;
                          DJOBCOUNT:=0;
                          PARWAIT(DINPORT,ID,2);
                          END;
                    2:BEGIN   (* JUST RECEIVED INPUT *)
                          DTIME:=MAX(DTIME,AMESSAGE.MTIME);
                          INCR(DJOBCOUNT);
                          DELAY:=TRUNC(-DEMU*LN(RANDOM(0))) + TRUNC(DCON);
                          IF (DCON=0.0) THEN INCR(DELAY);
                          AXQTTIME:=DELAY;
                          DSUMPTIME:=DSUMPTIME+DELAY;
                          DTIME:=DTIME+DELAY;
                          AMESSAGE.MTIME:=DTIME;
                          PARWAIT(DOUTPORT,ID,3);
                          END;
                    3:BEGIN   (* JUST SENT OUTPUT  *)
                          PARWAIT(DINPORT,ID,2);
                          END;
                    1000:BEGIN    (* LAST CALL, PRINT REPORT  *)
```

```
                 WRITE(TTY,' REPORT FROM');
                 PRSIGNATURE(ID);
                 WRITELN(TTY);
                 SHOWOWN(ID);
                 END;
            1010:BEGIN    (* W-SUB IJ COMPUTATION    *)
                 IF AWAITS(ID,DOUTPORT) THEN BEGIN
                    PORTS[DOUTPORT].PWIJ:=DTIME;
                    END ELSE BEGIN   (* AWAITING INPUT  *)
                       PORTS[DOUTPORT].PWIJ:=PORTS[DINPORT].PWIJ;
                       END;
                 END;
            1020:;  (* NO CHANGE OF AWAITED PORTS CAN OCCUR  *)
            OTHERS:BEGIN
                 PRSIGNATURE(ID);
                 WRITELN(TTY,' CALLED WITH BAD AMPC = ',AMPC:4);
                 SHOWPROCESS(ID);
                 SHOWOWN(ID);
                 IF (INTERTRACE>20) OR (TARGETTRACE>20) THEN DUMPTTY;
                 END;
            END;        (* CASE *)
         END;
   IF (INTERTRACE>0) OR (TARGETTRACE>0) THEN BEGIN
      WRITE(TTY,' LEAVING ');
      TRACELP(ID);
      END;
END;  (* LPDELAY PROCEDURAL DESCRIPTION *)




PROCEDURE LPMERGE2(ID:PROCID);
(* THE TYPE OF THIS LOGICAL PROCESS IS OBTAINED BY REMOVING THE
   'LP' FROM ITS NAME.  THIS DETERMINES THE OWNDATA ENTRY FOR
   PROCESSES OF THE TYPE DESCRIBED BY THE FOLLOWING CODE.     *)

PROCEDURE DECIDENEXT(ID:PROCID);
(* DECIDES THE NEXT AWAITED LINES FOR MERGE2
   ACCORDING TO THE FOLLOWING ACTION TABLE

   * HAVE1 * HAVE2 * IN1TIME<IN2TIME * IN1TIME=IN2TIME * IN2TIME<IN1TIME
   *********************************************************************
   * FALSE * FALSE * AWAIT BOTH INPT * AWAIT BOTH INPT * AWAIT BOTH
   * FALSE *  TRUE * WAIT 1 INPUT    * WAIT JOB2 OUT   * WAIT JOB2 OUT
   *  TRUE * FALSE * WAIT JOB1 OUT   * WAIT JOB1 OUT   * WAIT 2 INPUT
   *  TRUE *  TRUE * WAIT JOB1 OUT   * WAIT JOB2** OUT * WAIT JOB2 OUT

   THE ** ENTRY COULD ALSO BE WAIT JOB1 OUT.
   IT IS ASSUMED THAT ALL PREVIOUSLY SCHEDULED OUTPUT HAS BEEN SENT
   OUT.                                                                  *)
BEGIN
   WITH OWN[ID] DO
   WITH LPS[ID] DO
   IF NOT (M2HAVE1 OR M2HAVE2) THEN BEGIN
      (* AWAIT INPUT FROM BOTH INPUT PORTS, 1 AND 2 *)
      PARWAIT(M2IN1PORT,ID,2);
      PARWAIT(M2IN2PORT,ID,3);
      END ELSE
         IF (NOT M2HAVE1 AND (M2IN1TIME<M2IN2TIME)) THEN BEGIN
```

```
            (* AWAIT INPUT FROM PORT 1  *)
            PARWAIT(M2IN1PORT,ID,2);
            END ELSE IF (M2IN1TIME<M2IN2TIME) THEN BEGIN
               (* SEND MESSAGE FROM PORT 1  *)
               M2HAVE1:=FALSE;
               COPYMSGTOFROM(AMESSAGE,M2IN1MSG);
               AMESSAGE.MTIME:=MIN(M2IN1TIME,M2IN2TIME);
               PARWAIT(M2OUTPORT,ID,4);
               END ELSE IF M2HAVE2 THEN BEGIN
                  (* SEND MESSAGE FROM PORT 2  *)
                  M2HAVE2:=FALSE;
                  COPYMSGTOFROM(AMESSAGE,M2IN2MSG);
                  AMESSAGE.MTIME:=MIN(M2IN1TIME,M2IN2TIME);
                  PARWAIT(M2OUTPORT,ID,4);
                  END ELSE IF (M2IN1TIME=M2IN2TIME) THEN BEGIN
                     (* SEND MESSAGE FROM PORT 1  *)
                     M2HAVE1:=FALSE;
                     COPYMSGTOFROM(AMESSAGE,M2IN1MSG);
                     AMESSAGE.MTIME:=MIN(M2IN1TIME,M2IN2TIME);
                     PARWAIT(M2OUTPORT,ID,4);
                     END ELSE BEGIN
                        (* AWAIT INPUT FROM 2 *)
                        PARWAIT(M2IN2PORT,ID,3);
                        END;
END;  (* DECIDENEXT PROCEDURE...USED ONLY BY MERGE2 *)

BEGIN
   IF(TARGETTRACE>0) OR (INTERTRACE>0) THEN BEGIN
      WRITE(TTY,' (RE)-ENTERING ');
      PRSIGNATURE(ID);WRITELN(TTY);END;
   WITH LPS[ID] DO
      WITH OWN[ID] DO BEGIN
         AXQTTIME:=1;
         CASE AMPC OF
            0:BEGIN      (* CREATE THIS INSTANCE *)
                 WRITE(TTY,' CREATE');
                 PRSIGNATURE(ID);
                 WRITELN(TTY);
                 WRITE(TTY,' FOR INPUT PORT 1,');
                 DEFINEPORT(INN,ID,M2IN1PORT);
                 WRITE(TTY,' FOR INPUT PORT 2,');
                 DEFINEPORT(INN,ID,M2IN2PORT);
                 WRITE(TTY,' FOR OUTPUT PORT,');
                 DEFINEPORT(OUT,ID,M2OUTPORT);
                 END;
            1:BEGIN     (* FIRST CALL      *)
                 M2IN1TIME:=0;
                 M2JOB1COUNT:=0;
                 M2JOB2COUNT:=0;
                 M2IN2TIME:=0;
                 M2HAVE1:=FALSE;
                 M2HAVE2:=FALSE;
                 M2IN1MSG.MKIND:=JOB;
                 M2IN2MSG.MKIND:=JOB;
                 DECIDENEXT(ID);
                 END;
            2:BEGIN    (* JUST RECEIVED OVER IN1 *)
                 M2HAVE1:=TRUE;
                 M2IN1TIME:=AMESSAGE.MTIME;
```

```
               INCR(M2JOB1COUNT);
               COPYMSGTOFROM(M2IN1MSG,AMESSAGE);
               DECIDENEXT(ID);
               END;
          3:BEGIN     (* JUST RECEIVED OVER IN2 *)
               M2HAVE2:=TRUE;
               M2IN2TIME:=AMESSAGE.MTIME;
               INCR(M2JOB2COUNT);
               COPYMSGTOFROM(M2IN2MSG,AMESSAGE);
               DECIDENEXT(ID);
               END;
          4:BEGIN     (* JUST SENT OVER OUTPUT   *)
               DECIDENEXT(ID);
            END;
          1000:BEGIN    (* PRINT FINAL REPORT    *)
               WRITE(TTY,' REPORT FROM');
               PRSIGNATURE(ID);
               WRITELN(TTY);
               SHOWOWN(ID);
               END;
          1010:BEGIN    (* W-SUB IJ COMPUTATION *)
               IF AWAITS(ID,M2OUTPORT) THEN  (* WAITING TO OUTPUT *)
                  WITH PORTS[M2OUTPORT] DO
                     PWIJ:=MIN(M2IN1TIME,M2IN2TIME)
                        ELSE PORTS[M2OUTPORT].PWIJ:=
                           MIN( PORTS[M2IN1PORT].PWIJ,
                                PORTS[M2IN2PORT].PWIJ));
               END; (* W SUB IJ COMPUTATION *)
          1020:BEGIN     (* ATTEMPT TO CHANGE LINES AWAITED    *)
               IF NOT AWAITS(ID,M2OUTPORT) THEN BEGIN
                  (* MOVE LINE TIMES FORWARD IF POSSIBLE    *)
                  M2IN1TIME:=MAX(M2IN1TIME,PORTS[M2IN1PORT].PWIJ);
                  M2IN2TIME:=MAX(M2IN2TIME,PORTS[M2IN2PORT].PWIJ);
                  (* TRY FOR A DIFFERENT SET OF LINES       *)
                  PORTS[M2IN1PORT].PRELIGIBLE:=FALSE;
                  PORTS[M2IN2PORT].PRELIGIBLE:=FALSE;
                  DECIDENEXT(ID);
                  PORTS[M2IN1PORT].PRWAITING:=TRUE;
                  PORTS[M2IN2PORT].PRWAITING:=TRUE;
                  PORTS[M2OUTPORT].PSWAITING:=TRUE;
                  END;
               END;

          OTHERS:BEGIN
               PRSIGNATURE(ID);
               WRITELN(TTY,' CALLED WITH BAD AMPC = ',AMPC:4);
               SHOWPROCESS(ID);
               SHOWOWN(ID);
               IF (INTERTRACE>20) OR (TARGETTRACE>20) THEN DUMPTTY;
               END;
          END;        (* CASE *)
        END;
   IF (INTERTRACE>0) OR (TARGETTRACE>0) THEN BEGIN
      WRITE(TTY,' LEAVING ');
      TRACELP(ID);
      END;
END;  (* LPMERGE2 PROCEDURAL DESCRIPTION *)
```

```
PROCEDURE LPFORK2(ID:PROCID);
(* THE TYPE OF THIS LOGICAL PROCESS IS OBTAINED BY REMOVING THE
    'LP' FROM ITS NAME.  THIS DETERMINES THE OWNDATA ENTRY FOR
    PROCESSES OF THE TYPE DESCRIBED BY THE FOLLOWING CODE.     *)


BEGIN
    IF(TARGETTRACE>0) OR (INTERTRACE>0) THEN BEGIN
        WRITE(TTY,' (RE)-ENTERING ');
        PRSIGNATURE(ID);WRITELN(TTY);END;
    WITH LPS[ID] DO
        WITH OWN[ID] DO BEGIN
            AXQTTIME:=1;
            CASE AMPC OF
                0:BEGIN  (* CALL TO CREATE THE PROCESS *)
                      WRITE(TTY,' CREATE');
                      PRSIGNATURE(ID);
                      WRITELN(TTY);
                      WRITE(TTY,' FOR INPUT PORT,');
                      DEFINEPORT(INN,ID,F2INPORT);
                      WRITE(TTY,' FOR OUTPUT PORT 1,');
                      DEFINEPORT(OUT,ID,F2OUT1PORT);
                      WRITE(TTY,' FOR OUTPUT PORT 2,');
                      DEFINEPORT(OUT,ID,F2OUT2PORT);
                      WRITE(TTY,' WHAT IS PROBABILITY OF A BRANCH ');
                      WRITELN(TTY,'TO OUTPUT PORT 1 ?');
                      BREAK;
                      RESET(TTY);
                      READ(TTY,F2RHO);
                      END;
                1:BEGIN   (* FIRST CALL TO THIS INSTANCE OF FORK2 *)
                      F2TIME:=0;
                      F2OUT1COUNT:=0;
                      F2OUT2COUNT:=0;
                      PARWAIT(F2INPORT,ID,2);
                      END;
                2:BEGIN   (* JUST RECEIVED INPUT OVER F2INPORT *)
                      F2TIME:=AMESSAGE.MTIME;
                      IF(RANDOM(0)<=F2RHO) THEN BEGIN (* SEND ON 1 *)
                          INCR(F2OUT1COUNT);
                          PARWAIT(F2OUT1PORT,ID,3);
                          END ELSE BEGIN             (* SEND ON 2 *)
                              INCR(F2OUT2COUNT);
                              PARWAIT(F2OUT2PORT,ID,3);
                              END;
                      END;
                3:BEGIN   (* JUST SENT A JOB OUT *)
                      PARWAIT(F2INPORT,ID,2);
                      END;
                1000:BEGIN  (* FINAL CALL FOR REPORTS *)
                      WRITE(TTY,' REPORT FROM');
                      PRSIGNATURE(ID);
                      WRITELN(TTY);
                      SHOWOWN(ID);
                      END;

                1010:BEGIN   (* W SUB IJ COMPUTATION  *)
                      IF AWAITS(ID,F2INPORT) THEN BEGIN (* WAITING INPUT *)
                          PORTS[F2OUT1PORT].PWIJ:=PORTS[F2INPORT].PWIJ;
```

```
                     PORTS[F2OUT2PORT].PWIJ:=PORTS[F2INPORT].PWIJ;
                     END ELSE IF AWAITS(ID,F2OUT1PORT) THEN BEGIN
                     (* AWAITING OUTPUT ON OUTPUT PORT 1 *)
                     PORTS[F2OUT1PORT].PWIJ:=F2TIME;
                     PORTS[F2OUT2PORT].PWIJ:=PORTS[F2INPORT].PWIJ;
                     END ELSE BEGIN  (* AWAITING OUTPUT OVER PORT 2 *)

                     PORTS[F2OUT2PORT].PWIJ:=F2TIME;
                     PORTS[F2OUT1PORT].PWIJ:=PORTS[F2INPORT].PWIJ;
                     END;
                  END;   (* W SUB IJ COMPUTATION *)
             10201;      (* AFTER W SUB IJ COMPUTATION, FORK AWAITS THE
                            SAME PORTS IT DID BEFORE                    *)


             OTHERS:BEGIN
                  PRSIGNATURE(ID);
                  WRITELN(TTY,' CALLED WITH BAD AMPC = ',AMPC:4);
                  SHOWPROCESS(ID);
                  SHOWOWN(ID);
                  IF (INTERTRACE>20) OR (TARGETTRACE>20) THEN DUMPTTY;
                  END;
             END;        (* CASE *)
          END;
    IF (INTERTRACE>0) OR (TARGETTRACE>0) THEN BEGIN
       WRITE(TTY,' LEAVING ');
       TRACELP(ID);
       END;
END;  (* LPFORK2 PROCEDURAL DESCRIPTION *)


PROCEDURE LPQUEUE20(ID:PROCID);
(* THE TYPE OF THIS LOGICAL PROCESS IS OBTAINED BY REMOVING THE
   'LP' FROM ITS NAME,  THIS DETERMINES THE OWNDATA ENTRY FOR
   PROCESSES OF THE TYPE DESCRIBED BY THE FOLLOWING CODE,    *)

VAR I:INTEGER;

BEGIN
   IF(TARGETTRACE>0) OR (INTERTRACE>0) THEN BEGIN
      WRITE(TTY,' (RE)-ENTERING ');
      PRSIGNATURE(ID);WRITELN(TTY);END;
   WITH LPS[ID] DO
      WITH OWN[ID] DO BEGIN
         AXQTIME:=1;
         CASE AMPC OF
            0:BEGIN  (* CREATE *)
                 WRITE(TTY,' CREATE');
                 PRSIGNATURE(ID);
                 WRITELN(TTY);
                 WRITE(TTY,' FOR INPUT PORT,');
                 DEFINEPORT(INN,ID,Q20INPORT);
                 WRITE(TTY,' FOR OUTPUT PORT,');
                 DEFINEPORT(OUT,ID,Q20OUTPORT);
                 WRITELN(TTY,' ENTER QUEUE CAPACITY, 1..20 ');
                 BREAK;
                 RESET(TTY);
                 READ(TTY,Q20QMAX);
                 Q20QMAX:=Q20QMAX-1;
```

```
            Q20QMAX:=MAX(0,Q20QMAX);
            Q20QMAX:=MIN(19,Q20QMAX);
            END;
        1:BEGIN  (* FIRST CALL TO THIS INSTANCE OF A QUEUE20 *)
            Q20INPTR:=1;
            Q20OUTPTR:=1;
            Q20TIME:=0;
            Q20JOBCOUNT:=0;
            PARWAIT(Q20INPORT,ID,2);
            END;
        2:BEGIN  (* JUST RECEIVED OVER INPORT *)
            Q20TIME:=AMESSAGE.MTIME;
            INCR(Q20JOBCOUNT);
            Q20BUFFER[Q20INPTR MOD 20]:=AMESSAGE.MTIME;
            Q20BUFFER2[Q20INPTR MOD 20]:=AMESSAGE.JOBNUMBER;
            INCR(Q20INPTR);
            IF (Q20INPTR<=Q20OUTPTR+Q20QMAX) THEN
               PARWAIT(Q20INPORT,ID,2);
            AMESSAGE.MTIME:=Q20TIME;
            PARWAIT(Q20OUTPORT,ID,3);
            END;
       3:BEGIN   (* JUST SENT OVER OUTPORT  *)
            INCR(Q20OUTPTR);
            IF (Q20OUTPTR<Q20INPTR) THEN BEGIN
               AMESSAGE.MTIME:=Q20TIME;
               AMESSAGE.MKIND:=JOB;
               AMESSAGE.JOBNUMBER:=Q20BUFFER2[Q20OUTPTR MOD 20];
               PARWAIT(Q20OUTPORT,ID,3);
               END;
            PARWAIT(Q20INPORT,ID,2);
            END;
      1000:BEGIN   (* FINAL CALL FOR REPORT  *)
            WRITE(TTY,' REPORT FROM');
            PRSIGNATURE(ID);
            WRITELN(TTY);
            SHOWOWN(ID);
            END;
      1010:BEGIN   (* W SUB IJ COMPUTATION   *)
            IF AWAITS(ID,Q20OUTPORT) THEN BEGIN  (* WAIT OUTPUT *)
               PORTS[Q20OUTPORT].PWIJ:=Q20TIME;
               END ELSE BEGIN
                  PORTS[Q20OUTPORT].PWIJ:=PORTS[Q20INPORT].PWIJ;
                  END;
            END;
        1020:;    (* A QUEUE CANNOT CHANGE AWAITED LINES *)


        OTHERS:BEGIN
            PRSIGNATURE(ID);
            WRITELN(TTY,' CALLED WITH BAU AMPC = ',AMPC:4);
            SHOWPROCESS(ID);
            SHOWOWN(ID);
            IF (INTERTRACE>20) OR (TARGETTRACE>20) THEN DUMPTTY;
            END;
        END;        (* CASE *)
     END;
IF (INTERTRACE>0) OR (TARGETTRACE>0) THEN BEGIN
   WRITE(TTY,' LEAVING ');
   TRACELP(ID);
```

```
       END;
END;  (* LPQUEUE20 PROCEDURAL DESCRIPTION *)

(* HERE IS A BABY LP TO PLAY WITH ....


PROCEDURE LPNAME(ID);
(* THE TYPE OF THIS LOGICAL PROCESS IS OBTAINED BY REMOVING THE
    'LP' FROM ITS NAME.  THIS DETERMINES THE OWNDATA ENTRY FOR
    PROCESSES OF THE TYPE DESCRIBED BY THE FOLLOWING CODE.    ENDCOMMENT

VAR

BEGIN
    IF(TARGETTRACE>0) OR (INTERTRACE>0) THEN BEGIN
       WRITE(TTY,' (RE)=ENTERING ');
       PRSIGNATURE(ID);WRITELN(TTY);END;
    WITH LPS[ID] DO
       WITH OWN[ID] DO BEGIN
           AXQTTIME:=1;
           CASE AMPC OF
              0:BEGIN
                   PRSIGNATURE(ID);
                   END;




              OTHERS:BEGIN
                   PRSIGNATURE(ID);
                   WRITELN(TTY,' CALLED WITH BAD AMPC = ',AMPC:4);
                   SHOWPROCESS(ID);
                   SHOWOWN(ID);
                   IF (INTERTRACE>20) OR (TARGETTRACE>20) THEN DUMPTTY;
                   END;
              END;       (* CASE ENDCOMMENT
           END;
    IF (INTERTRACE>0) OR (TARGETTRACE>0) THEN BEGIN
       WRITE(TTY,' LEAVING ');
       TRACELP(ID);
       END;
END;  (* LPNAME PROCEDURAL DESCRIPTION ENDCOMMENT


       END OF THE COMMENT CONTAINING THE PROCEDURAL TEMPLATES     *)






PROCEDURE INITIALIZE;
BEGIN
   NETDEADLOCK:=FALSE;
   NETTERM:=FALSE;
   NETSUMXQTTIME:=0;
   NETSUMCMNTIME:=0;
   NETSUMBLKTIME:=0;
   DEADLOCKCOUNT:=0;
```

```
      NETTIME:=0;
      NETFAIR:=1;
      NETMSGCOUNT:=0;
   END;

   PROCEDURE SETTRACE;
   (* SOLICITS VALUES FOR TRACE VARS AND TIME AND MESSAGE LIMITS
      FROM THE OPERATOR *)
   BEGIN
      WRITELN(TTY,' SET TRACE VALUES, 0==> NO TRACE, BIG==> MORE TRACE');
      WRITELN(TTY,' REENTER THE INTERPRETER TRACE VALUE');
      BREAK;
      RESET(TTY);
      READ(TTY,INTERTRACE);
      WRITELN(TTY,' ECHO ',INTERTRACE:4,' ENTER TARGET TRACE ');
      BREAK;
      RESET(TTY);
      READ(TTY,TARGETTRACE);
      WRITELN(TTY,' ECHO ',TARGETTRACE:4,' ENTER DEADLOCK TRACE ');
      BREAK;
      RESET(TTY);
      READ(TTY,NETDEADTRACE);
      WRITELN(TTY,' ECHO ',NETDEADTRACE:4,' ENTER COMMUNICATION TRACE');
      BREAK;
      RESET(TTY);
      READ(TTY,NETCMNTRACE);
      WRITELN(TTY,' ECHO ',NETCMNTRACE:3,' ENTER EXECUTION TRACE');
      BREAK;
      RESET(TTY);
      READ(TTY,NETXQTTRACE);
      WRITELN(TTY,' ECHO ',NETXQTTRACE:3,' ENTER BLOCKING TRACE');
      BREAK;
      RESET(TTY);
      READ(TTY,NETBLKTRACE);
      WRITE(TTY,' ECHO ',NETBLKTRACE:3,' MESSAGE COUNT= ',NETMSGCOUNT);
      WRITELN(TTY,' ENTER MESSAGE LIMIT');
      BREAK;
      RESET(TTY);
      READ(TTY,NETMSGLIMIT);
      WRITELN(TTY,' ECHO ',NETMSGLIMIT,' TIME NOW = ',NETTIME);
      WRITELN(TTY,' ENTER TIME LIMIT');
      BREAK;
      RESET(TTY);
      READ(TTY,NETTIMELIMIT);
      WRITE(TTY,' ECHO ',NETTIMELIMIT:6,' THERE HAVE BEEN ');
      WRITELN(TTY,DEADLOCKCOUNT:4,' DEADLOCKS, ENTER NEW LIMIT');
      BREAK;
      RESET(TTY);
      READ(TTY,DEADLOCKLIMIT);
      WRITELN(TTY,' ECHO ',DEADLOCKLIMIT:4,' END SETTRACE ');
   END;


   PROCEDURE RESUME(PROCNUM:PROCID);

   (* RESUME IS INVOKED WHEN THE PROCESS PROCNUM ENTERS THE EXECUTING
      STATE.  PROCNUM ALSO INDEXES THE ARRAYS LPS AND OWN CONTAINING
      THE PROCESS' STATE AND OWNED DATA RESPECTIVELY.              *)

```

```
BEGIN
   IF (INTERTRACE+NETCMNTRACE+NETXQTTRACE+TARGETTRACE) > 0 THEN
      WRITELN(TTY,' PROCESS ',PROCNUM:3,' RESUMED');
   LPS[PROCNUM].ASTATE := XQT;
   LPS[PROCNUM].ANEXTSTATE := TRM; (* PROCESS MAY SET TO BLK WITH PARWAIT *)
(* IN W-SUB IJ COMPUTATION, PROCESSES REMAIN BLOCKED UNLESS
   IN THE PROCESS OF COMPUTATION, A PROCESS DETERMINES ITS NEXT
   STATE WILL BE TERMINATED, AND SETS ITS ANEXTSTATE ACCORDINGLY  *)
   IF (LPS[PROCNUM].AMPC=1010) OR (LPS[PROCNUM].AMPC=1020) THEN BEGIN
      LPS[PROCNUM].ASTATE:=BLK;
      LPS[PROCNUM].ANEXTSTATE:=BLK;
      END;
   CASE LPS[PROCNUM].ATYPE OF

   (* TARDEP * RESUME PROCESS PROCEDURE'S PROGRAM *)

      SOURCE:LPSOURCE(PROCNUM);
      SINK:LPSINK(PROCNUM);
      FORK2:LPFORK2(PROCNUM);
      MERGE2:LPMERGE2(PROCNUM);
      DELAY:LPDELAY(PROCNUM);
      QUEUE20:LPQUEUE20(PROCNUM);
      END;
   WITH LPS[PROCNUM] DO BEGIN
      ATIMELEFT := AXQTTIME;
      END;
END;

PROCEDURE FIREPORT(PORTNUM:PORTID);

(* THE PORT PORTNUM IS COMMITTED TO FIRE.  THE MESSAGE IS COPIED
   FROM THE PORT MESSAGE BUFFER INTO THE RECEIVER'S BUFFER.  THE
   SENDING AND RECEIVING PROCESSES ARE MARKED AS STATE = COM-
   MUNICATING, AND THEIR TIME REMAINING FIELDS GET THE SENDTIME
   AND RECEIVETIME SPECIFIC TO THIS PORT.  THE MESSAGE COUNTS ARE
   INCREMENTED FOR THE PORT, AND NETWORK TOTAL.  THE SENDER AND
   RECEIVER ARE DISQUALIFIED FROM FURTHER COMMUNICATION BY SETTING
   APPROPRIATE ELIGIBLE FIELDS FOR ALL PORTS NAMING THEM.  THE
   PROCESSES WILL NOT BE ACTUALLY RESUMED UNTIL THEY HAVE WAITED OUT
   THEIR COMMUNICATION TIME.                                       *)

VAR I:PORTID;
BEGIN
   IF (INTERTRACE+NETBLKTRACE+NETCMNTRACE+TARGETTRACE) > 0 THEN BEGIN
      WRITELN(TTY);
      WRITELN(TTY,' PORT ',PORTNUM:3,' FIRED');
      END;
   IF (INTERTRACE>10) OR (NETBLKTRACE>10)
      OR (NETCMNTRACE>10) OR (TARGETTRACE>10) THEN
      SHOWMSG(PORTS[PORTNUM].PMESSAGE);
   INCR(NETMSGCOUNT);
   INCR(PORTS[PORTNUM].PMSGCOUNT);
   WITH PORTS[PORTNUM] DO BEGIN
      WITH LPS[PSENDER] DO BEGIN
         IF (INTERTRACE>10) OR (NETBLKTRACE>10)
            OR (NETCMNTRACE>10) OR (TARGETTRACE>10) THEN BEGIN
            WRITE(TTY,' PROCESS ',PSENDER:3,' GOES FROM ');
            WRITELN(TTY,' BLK TO CMN AS SENDER');
            END;
```

```
         AMPC := PSMPC;
         ATIMELEFT := PSTIME;
         ASTATE := CMN;
         APORT := PORTNUM;
         END;
      WITH LPS[PRECEIVER] DO BEGIN
         IF (INTERTRACE>10) OR (NETBLKTRACE>10)
            OR (NETCMNTRACE>10) OR (TARGETTRACE>10) THEN BEGIN
            WRITE(TTY,' PROCESS ',PRECEIVER:3,' GOES FROM');
            WRITELN(TTY,' BLK TO CMN AS RECEIVER');
            END;
         ATIMELEFT := PRTIME;
         ASTATE := CMN;
         APORT := PORTNUM;
         AMPC := PRMPC;
         COPYMSGTOFROM(AMESSAGE,PMESSAGE);
         END;
      END;
   (* TURN OFF ELIGIBLE BITS FOR PORTS NAMING PROCESS I    *)
   FOR I:=1 TO HIGHPORT DO BEGIN
      PORTS[I].PSELIGIBLE:=(PORTS[I].PSELIGIBLE) AND
         (PORTS[I].PSENDER<>PORTS[PORTNUM].PSENDER);
      PORTS[I].PRELIGIBLE:=(PORTS[I].PRELIGIBLE) AND
         (PORTS[I].PRECEIVER<>PORTS[PORTNUM].PRECEIVER);
      END;
END;


PROCEDURE SCHEDULE(VAR PORTFIRINGORDER:PORTPOINTERS);

(* USER MAY PROVIDE THE SCHEDULER PROCEDURE TO CHANGE THE WAY THE
   PORTS ARE SELECTED TO FIRE.  AT END, PORTFIRINGORDER[I]=0 IFF
   THERE ARE NOT AS MANY AS I READY PORTS, AND PORTFIRINGORDER[I]=J
   IFF J IS TO BE THE ITH PORT TO FIRE.
   THIS IMPLEMENTATION IS FAIR BECAUSE IT FAVORS THE PORT = NETFAIR
   IF THIS PORT IS READY TO FIRE, AND NETFAIR IS ALWAYS INCREMENTED
   MODULO HIGHPORT WHEN PROCEDURE SCHEDULE IS INVOKED.  HENCE HIGHPORT
   IS AN UPPER LIMIT TO THE TIME THAT A READY PORT CAN WAIT FOR
   COMMUNICATION.  NO PROCESS IS SCHEDULED FOR
   MORE THAN ONE MESSAGE FIRING.
                                                                    *)

VAR  OKFIRE:ARRAY[1..PROCMAX] OF BOOLEAN;NEXT,PROCI:PROCID;PORTJ:PORTID;


BEGIN
IF (INTERTRACE>0)OR(NETCMNTRACE>0) THEN WRITELN(TTY,' SCHEDULE CALLED');
   IF (INTERTRACE>30) OR (NETCMNTRACE>30) THEN BEGIN
      WRITELN(TTY,' HERE ARE THE PORTS');
      FOR PORTJ:=1 TO HIGHPORT DO SHOWPORT(PORTJ);
      END;
   FOR PORTJ:=1 TO PORTMAX DO PORTFIRINGORDER[PORTJ]:=0;
   FOR PROCI:=1 TO PROCMAX DO OKFIRE[PROCI]:=TRUE;
   NEXT := 1;
   WITH PORTS[NETFAIR] DO
      IF PRWAITING AND PSWAITING AND PRELIGIBLE AND PSELIGIBLE THEN BEGIN
         PORTFIRINGORDER[NEXT]:=NETFAIR;
         NEXT:=2;
         IF NETFAIR=HIGHPORT THEN NETFAIR:=1 ELSE INCR(NETFAIR);
```

```
         OKFIRE[PSENDER]:=FALSE;
         OKFIRE[PRECEIVER]:=FALSE;
         END;
   FOR PORTJ:=1 TO HIGHPORT DO WITH PORTS[PORTJ] DO
      IF PRWAITING AND PSWAITING AND PRELIGIBLE AND PSELIGIBLE
         AND OKFIRE[PSENDER] AND OKFIRE[PRECEIVER] THEN BEGIN
            PORTFIRINGORDER[NEXT]:=PORTJ;
            INCR(NEXT);
            OKFIRE[PSENDER]:=FALSE;
            OKFIRE[PRECEIVER]:=FALSE;
            END;
END;


PROCEDURE PASSMESSAGES;

(* SCAN THE COMMUNICATIONS TABLES, IN THE GLOBAL PORTS, AND
   SELECT THE PORTS THAT WILL FIRE DURING THE CURRENT MOMENT
   OF TIME, AND FIRE THEM.  PASSMESSAGES HANDLES THINGS PORTS DO  *)

VAR PORTSTOBEFIRED:ARRAY[1..PORTMAX] OF INTEGER;NEXT:INTEGER;

BEGIN
   IF (INTERTRACE>0) OR (NETCMNTRACE>0) THEN
      WRITELN(TTY,' PASSMESSAGES CALLED');
   SCHEDULE(PORTSTOBEFIRED);
   NEXT := 1;
   WHILE (PORTSTOBEFIRED[NEXT] <> 0) AND (NEXT <= HIGHPORT) DO BEGIN
      FIREPORT(PORTSTOBEFIRED[NEXT]);
      INCR(NEXT);
      END;
END;

PROCEDURE AXE(PROCI:PROCID);
(* AXE CAN BE CALLED TO MARK A PROCESS AS TERMINATED FOR ANY
   ABNORMAL TERMINATION NOT DICTATED BY THE LOGICAL PROCESS CODE *)

VAR PORTJ:PORTID;
BEGIN
   IF (INTERTRACE+NETXQTTRACE+NETTRMTRACE+TARGETTRACE) > 0 THEN BEGIN
      WRITE(TTY,' AXE CALLED TO TERMINATE PROCESS ',PROCI:3);
      WRITELN(TTY);
      END;
   WITH LPS[PROCI] DO BEGIN
      ASTATE:=TRM;
      FOR PORTJ:=1 TO HIGHPORT DO WITH PORTS[PORTJ] DO BEGIN
         (* IF PORTJ NAMES PROCI, SET FALSE THE READY AND ELIGIBLES *)
         PSWAITING:=PSWAITING AND (PSENDER<>PROCI);
         PRWAITING:=PRWAITING AND (PRECEIVER<>PROCI);
         PSELIGIBLE:=PSELIGIBLE AND (PSENDER<>PROCI);
         PRELIGIBLE:=PRELIGIBLE AND (PRECEIVER<>PROCI);
         END;
      END;
END;


PROCEDURE TICK;
(* TICK UPDATES THE GLOBAL CLOCK NETTIME, INCREMENTING IT ONCE
   PER CALL.  ALSO, CHARGE ALL PROCESSES WITH ONE TIME UNIT
```

```
    ACCORDING TO THEIR STATE.  PROCESSES WITH ASTATE= CMN OR XQT
    ARE COUNTING DOWN THEIR ATIMELEFT FIELDS.  IF IT NOW BECOMES
    ZERO, ENTER STATES BLK OR TRM, OR XQT RESPECTIVELY.  A PROCESS
    ENTERING THE BLK STATE HAS ALREADY SPECIFIED THE PORTS OVER
    WHICH IT IS ELIGIBLE TO COMMUNICATE VIA PARWAIT CALLS.  MARK
    THEM AS XWAITING NOW, X=S SENDER OR R RECEIVER.    TICK BASICALLY
    SEES ALL OF THE PROCESSES THROUGH THE CURRENT TIME UNIT, WHATEVER
    THEIR CURRENT STATE.  TICK HANDLES THINGS PROCESSES DO      *)

VAR PROCI:PROCID;PORTJ:PORTID;
    ALIVE:BOOLEAN;

BEGIN
    IF INTERTRACE#0 THEN WRITELN(TTY,' TICK CALLED, NETTIME = ',NETTIME:7);
    INCR(NETTIME);
    FOR PROCI:=1 TO HIGHPROC DO
        WITH LPS[PROCI] DO BEGIN
            CASE ASTATE OF
                XQT : BEGIN
                           IF (INTERTRACE>20) OR (NETXQTTRACE>20)
                               OR (TARGETTRACE>20) THEN BEGIN
                               WRITE(TTY,' PROCESS ',PROCI:3,' IS EXECUTING ');
                               WRITELN(TTY,'TIMELEFT IS ',ATIMELEFT:3);
                               END;
                           DECR(ATIMELEFT);
                           INCR(ASUMXQTTIME);
                           INCR(NETSUMXQTTIME);
                           IF ATIMELEFT<=0 THEN BEGIN
                               (*  POSSIBLE NEXT STATES ARE TRM AND BLK    *)
                               ASTATE:=ANEXTSTATE;
                               IF ASTATE=TRM THEN BEGIN
                                   ATRMTIME:=NETTIME;
                                   IF (INTERTRACE>10) OR (NETXQTTRACE>10)
                                       OR (NETTRMTRACE>10) OR (TARGETTRACE>10)
                                       THEN BEGIN
                                       WRITE(TTY,' PROCESS ',PROCI:3,' GOES ');
                                       WRITELN(TTY,'FROM XQTING TO TRMED');
                                       END;
                                   END ELSE   (* ASTATE = BLK  *)
                                       IF (INTERTRACE>10) OR (NETXQTTRACE>10)
                                          OR (NETBLKTRACE>10)
                                              OR (TARGETTRACE>10) THEN BEGIN
                                          WRITE(TTY,' PROCESS ',PROCI:3,' GOES');
                                          WRITELN(TTY,' FROM XQTING TO BLKED');
                                          END;
                               END;  (* IF ATIMELEFT<=0 *)
                           END;  (* PROCI XQTING  *)
                CMN :BEGIN
                           IF (INTERTRACE>20) OR (NETCMNTRACE>20)
                               OR (TARGETTRACE>20) THEN BEGIN
                               WRITE(TTY,' PROCESS ',PROCI:3,' IS COMMUNICA');
                               WRITELN(TTY,'TING, TIMELEFT IS ',ATIMELEFT:3);
                               END;
                           INCR(NETSUMCMNTIME);
                           DECR(ATIMELEFT);
                           INCR(ASUMCMNTIME);
                           IF(ATIMELEFT=0) THEN RESUME(PROCI);
                           END;
                BLK : BEGIN
```

```
                     IF (INTERTRACE>20) OR (NETBLKTRACE>20)
                        OR (TARGETTRACE>20) THEN BEGIN
                        WRITE(TTY,' PROCESS ',PROCI:3,' BLOCKED');
                        WRITELN(TTY);
                        END;
                     INCR(NETSUMBLKTIME);
                     INCR(ASUMBLKTIME);
                     END;
            TRM :;
            END;
         END;    (* WITH  *)
   FOR PROCI:=1 TO HIGHPROC DO
      IF (LPS[PROCI].ASTATE = BLK) THEN BEGIN
         (* ALL BLOCKED PROCESSES MUST BE WAITING FOR AT LEAST ONE
         PROCESS THAT HAS NOT TRMED                                *)
         ALIVE:=FALSE;   (* IF THERE IS NONE, THIS PROCESS IS AXED *)
         FOR PORTJ:=1 TO HIGHPORT DO WITH PORTS[PORTJ] DO BEGIN
            (* IF PORTJ NAMES PROCI THEN MARK THE APPR. XWAITING TRUE *)
            PSWAITING:=PSWAITING
                  OR (PSENDER=PROCI);
            PRWAITING:=PRWAITING
                  OR (PRECEIVER=PROCI);
            (* IF PROCI IS AWAITING A LIVE PARTNER ON THIS PORT, THEN *)
            ALIVE:=ALIVE OR      (* MARK IT AS LIVING *)
                   ((PRECEIVER=PROCI) AND (LPS[PSENDER].ASTATE<>TRM));
            ALIVE:=ALIVE OR
                  ((PSENDER=PROCI) AND (LPS[PRECEIVER].ASTATE<>TRM));
            END;
         IF NOT ALIVE THEN BEGIN
            AXE(PROCI);
            IF (INTERTRACE+NETBLKTRACE+NETTRMTRACE+TARGETTRACE) > 0 THEN
               WRITELN(TTY,' BECAUSE ALL AWAITED PROCESSES ARE TRMED');
            END;
         END;   (* IF BLKED *)
END;       (* TICK  *)

PROCEDURE COUNTBYSTATE(VAR RESULT:TALLEYBYSTATE);
(* COUNT THE NUMBER OF PROCESSES IN EACH STATE               *)
VAR PROCI:PROCID;
BEGIN
   IF INTERTRACE<>0 THEN WRITELN(TTY,' COUNTBYSTATE CALLED');
   WITH RESULT DO BEGIN
      XQTING:=0;
      CMNING:=0;
      BLKED :=0;
      TRMED :=0;
      FOR PROCI:=1 TO HIGHPROC DO
         CASE LPS[PROCI].ASTATE OF
            XQT:INCR(XQTING);
            BLK:INCR(BLKED);
            CMN:INCR(CMNING);
            TRM:INCR(TRMED);
            END;
      IF INTERTRACE>10 THEN BEGIN
         WRITE(TTY,' COUNTS ARE XQTING = ',XQTING:3,' CMNING = ');
         WRITELN(TTY,CMNING:3,' BLKED = ',BLKED:3,' TRMED = ',TRMED:3);
         END;
      END;  (* WITH *)
END;
```

```
PROCEDURE PRINTSTATISTICS;
(* CAN BE EXTENDED AS THE NEED ARISES                     *)
VAR PRUCI:PROCID;PORTJ:PORTID;SUM:INTEGER;FACTOR:REAL;
BEGIN
    WRITELN(TTY,' NETWORK PERFORMANCE STATISTICS FOLLOW:');
    WRITE(TTY,' ELAPSED TIME = ',NETTIME:7,' TOTAL EXECUTION TIME = ');
    WRITELN(TTY,NETSUMXQTTIME:8,' FOR A MULTIPROCESSING FACTOR ');
    FACTOR:=NETSUMXQTTIME/NETTIME;
    WRITE(TTY,'OF ',FACTOR,' TOTAL MESSAGE COUNT = ');
    WRITELN(TTY,NETMSGCOUNT:7,' TOTAL COMMUNICATION TIME =:');
    WRITE(TTY,NETSUMCMNTIME:6);
    WRITELN(TTY,' NETSUMBLKTIME=',NETSUMBLKTIME:6,' PORT SUMMARIES: ');
    WRITELN(TTY,' PORTID SENDER RECEIVER STIME RTIME MSG-COUNT');
    FUR PORTJ:=1 TO HIGHPORT DO WITH PORTS[PORTJ] DO BEGIN
       WRITE(TTY,' ',PORTJ:4,'  ',PSENDER:4,'       ',PRECEIVER:4);
       WRITELN(TTY,'     ',PSTIME:4,' ',PRTIME:4,'     ',PMSGCOUNT:5);
       END;
    IF (INTERTRACE >10) OR (TARGETTRACE>10) THEN BEGIN
       WRITELN(TTY,' PROCESS SUMMARY:');
       FOR PROCI:=1 TO HIGHPROC DO SHOWPROCESS(PROCI);
       WRITELN(TTY);
    END;
    WRITE(TTY,' THERE WERE ',DEADLOCKCOUNT:5,' DEADLOCKS IN');
    WRITELN(TTY,' THE SEQUENTIAL SIMULATION ');
END;

PROCEDURE INITPORTS;
(* TARDEP * INITIALIZE THE PORT TABLES                   *)
VAR I:PORTID;
BEGIN
    IF INTERTRACE<>0 THEN WRITELN(TTY,' INITPORTS CALLED');
    FOR I:=1 TO PORTMAX DO
       WITH PORTS[I] DO BEGIN
          PSELIGIBLE:=FALSE;
          PSWAITING:=FALSE;
          PRWAITING:=FALSE;
          PRELIGIBLE:=FALSE;
          PSMPC:=1;
          PRMPC:=1;
          PMSGCOUNT:=0;
          END;
    WRITELN(TTY,' ENTER THE TOTAL NUMBER OF PORTS USED IN THIS NET');
    BREAK;
    RESET(TTY);
    READ(TTY,HIGHPORT);
    WRITELN(TTY,' ECHO ',HIGHPORT:4);
END;

PROCEDURE INITPROCS;
(* TARDEP * INITIALIZE PROCEDURE TABLES                  *)
VAR I,INDEX,IDNEXT,KINDMAX:0..PROCMAX;KIND:PROCKIND;
BEGIN
    IF INTERTRACE<>0 THEN WRITELN(TTY,' INITPROCS CALLED');
    FOR I:=1 TO PROCMAX DO WITH LPS[I] DO BEGIN
       ASTATE:=XQT;
       ASUMXQTTIME:=0;
       ASUMBLKTIME:=0;
       ASUMCMNTIME:=0;
```

```
      ATRMTIME:=0;
      END;
   IDNEXT:=1;
   FOR KIND:= SOURCE TO QUEUE22 DO BEGIN
      WRITE(TTY,' HOW MANY PROCESSES OF TYPE ');
      PRPROCKIND(KIND);
      WRITELN(TTY,' ?');
      BREAK;
      RESET(TTY);
      READ(TTY,KINDMAX);
      IF (KINDMAX>0) THEN
         FOR I:=1 TO KINDMAX DO BEGIN
            OWN[IDNEXT].OPROCKIND:=KIND;
            LPS[IDNEXT].ATYPE:=KIND;
            LPS[IDNEXT].AINSTANCE:=I;
            LPS[IDNEXT].AMPC:=0;
            (* THAT ASSIGNMENT DISTINGUISHED THIS CALL AS A CREATE *)
            RESUME(IDNEXT);
            INCR(IDNEXT);
            END;
      END;
   HIGHPROC:=IDNEXT-1;
END;

PROCEDURE BREAKDEADLOCK;
(* ATTEMPT TO BREAK DEADLOCK BY LETTING ALL PROCESSES COMPUTE W SUB IJ
   BY RESUMING THEM AT AMPC = 1010.  EACH CALL TO PROCI LETS PROCI
   REVISE DOWNWARD OR LEAVE CONSTANT ITS ESTIMATE OF THE EARLIEST
   PROCI COULD TRY TO SEND ON THE ARC TO PROCJ.  WHEN PROCI IS
   SO RESUMED THE K'TH TIME, IT COMPUTES THE K'TH ESTIMATE OF
   W SUB IJ FOR ALL OUTPUT PORTS TO PROCJ BASED ON THE AVAILABILITY
   OF THE (K-1)TH W SUB HI ON ALL INPUT PORTS FROM PROCH, THIS
   HAVING BEEN COMPUTED ON THE PREVIOUS WIJPASS.
   PORTS HAVE A .PWIJ FIELD WHICH IS WRITTEN BY THE SENDER, AND
   READ BY THE RECEIVER, OF MESSAGES ON THE BOUND PORT.            *)


VAR WIJPASS,PROCI:PROCID;PORTP:PORTID;
BEGIN
   IF (INTERTRACE+NETDEADTRACE+NETCMNTRACE+NETXQTTRACE+NETBLKTRACE>0)
      THEN BEGIN
      WRITELN(TTY,' BREAKDEADLOCK ATTEMPTED AT NETTIME = ',NETTIME:8);
      END;
   FOR PROCI:= 1 TO HIGHPROC DO LPS[PROCI].AMPC:=1010;
   FOR PORTP:= 1 TO HIGHPORT DO WITH PORTS[PORTP] DO BEGIN
      PWIJ:=MAXINT;
      END;
   FOR WIJPASS:= 1 TO HIGHPROC DO BEGIN
      IF (NETDEADTRACE>10) THEN BEGIN
          WRITELN(TTY,' COMPUTE W (',WIJPASS:3,') FOR ALL PORTS');
          END;
      FOR PROCI:= 1 TO HIGHPROC DO BEGIN
         RESUME(PROCI);
         END;
      IF (NETDEADTRACE>20) THEN BEGIN
         WRITE(TTY,' HERE ARE W (K=',WIJPASS:3,') SUB IJ FOR   ');
         WRITELN(TTY,'ALL PORTS IJ ');
         WRITE(TTY,'          I         J          W(K) SUB IJ      ');
      WRITELN(TTY);
```

```
         FOR PORTP:=1 TO HIGHPORT DO WITH PORTS[PORTP] DO BEGIN
            WRITE(TTY,'      ',PSENDER:3,'      ',PRECEIVER:3);
            WRITELN(TTY,'                ',PWIJ:8,'    ');
            END;
         END;
      END;
   FOR PROCI:=1 TO HIGHPROC DO BEGIN
      LPS[PROCI].AMPC:=1020;
      RESUME(PROCI);
      END;
END;

(*           MAIN    PROGRAM          *)

BEGIN
   WRITELN(TTY,' BEGIN PROGRAM DSIM');
   INITPROCS ;          (* INIT TARGET NETWORK PROCESS  RECORDS *)
   INITPORTS;           (* INIT TARGET NETWORK PORT RECORDS     *)
   WRITELN(TTY,' END OF NETWORK SPECIFICATION, INSPECT AND VERIFY ');
   SHOWNETWORK;         (* DISPLAY NETWORK FOR USER          *)
   BUFFERSIZE:=1;       (* BUFFERSIZES WILL BE 1,2,4,8, & 16     *)
   REPEAT               (* RUN THE PROGRAM WITH A NEW BUFFERSIZE *)
   INITIALIZE;          (* INITIALIZE INTERPRETER VARS FOR FRESH RUN *)
   INITPORTS;           (* INIIALIZE THE PORT RECORDS                *)
   SETTRACE;            (* INITIALIZE THE TRACE AND LIMIT VARIABLES  *)
   WRITELN(TTY);
   WRITE(TTY,' BEGIN THE SIMULATION, NETTIME = ',NETTIME:8);
   WRITELN(TTY,' BUFFERSIZE = ',BUFFERSIZE:2);
   BREAK;
   FOR PROCI:=1 TO HIGHPROC DO BEGIN  (* ACTIVATE PROCESS I *)
      LPS[PROCI].AMPC:= 1;
                        (* THAT ASSIGNMENT DISTINGUISH THIS CALL TO
                           THE PROCESSES AS THEIR INITIAL ACTIVATION *)
      RESUME(PROCI);
                        (* FORCING THE QUEUE20 BUFFERSIZES FOR EXPT.  *)
      IF LPS[PROCI].ATYPE=QUEUE20 THEN OWN[PROCI].Q20QMAX:=BUFFERSIZE-1;
      END;              (* ALL PROCESSES ARE EXECUTING *)
   IF (INTERTRACE>50) THEN DUMPTTY;
   REPEAT               (* SIMULATE PASSAGE OF ONE TIME UNIT FOR NETWORK *)
      TICK;             (* ALL LPS DO THEIR THING FOR ONE TIME UNIT *)
      PASSMESSAGES;  (* FIRE SOME READY PORTS IF POSSIBLE *)
      COUNTBYSTATE(COUNT);    (* COUNT THE SURVIVORS, AND OTHERS    *)
      IF (COUNT.XQTING=0) AND (COUNT.CMNING=0) THEN  (* NO ONE ALIVE  *)
         IF (COUNT.TRMED>=HIGHPROC) THEN NETTERM:=TRUE  (* ALL DEAD *)
            ELSE BEGIN        (* NEITHER DEAD NOR ALIVE--  DEADLOCKED! *)
               IF(NETDEADTRACE>0) THEN BEGIN
                  WRITELN(TTY,' NET DEADLOCKED, NETTIME = ',NETTIME);
                  END;
               NETDEADLOCK:=TRUE;
               INCR(DEADLOCKCOUNT);
               BREAKDEADLOCK;
               PASSMESSAGES;
               COUNTBYSTATE(COUNT);
               NETDEADLOCK:= (COUNT.XQTING = 0) AND (COUNT.CMNING = 0);
               IF (INTERTRACE>0) OR (NETDEADTRACE>0)
                  OR (TARGETTRACE>0) THEN BEGIN
                     WRITE(TTY,' PROCEDURE BREAKDEADLOCK WAS ');
                     CASE NETDEADLOCK OF
                        TRUE:WRITELN(TTY,'UNABLE TO BREAK DEADLOCK');
```

```
                        FALSE:WRITELN(TTY,'ABLE TO BREAK DEADLOCK');
                        END;
                     END;

               END;
      (* IF TIME OR MESSAGE LIMITS EXCEEDED, GIVE OPERATOR A CHANCE
         TO ENTER NEW LIMITS, CHANGE THE TRACE VARIABLES, ETC.       *)
      IF (NETTIME>NETTIMELIMIT) OR (NETMSGCOUNT>NETMSGLIMIT)
         OR (INTERTRACE>30) OR (DEADLOCKCOUNT>DEADLOCKLIMIT) THEN BEGIN
            SETTRACE;
            WRITELN(TTY,' RESUME SIMULATION, WATCH YOUR TIME ');
            BREAK;
            END;
      UNTIL NETDEADLOCK OR NETTERM OR (NETTIME>NETTIMELIMIT)
         OR (NETMSGCOUNT>NETMSGLIMIT) OR (DEADLOCKCOUNT>DEADLOCKLIMIT);
   WRITE(TTY,' TARGET SIMULATION TERMINATED BECAUSE');
   IF (DEADLOCKCOUNT>DEADLOCKLIMIT) THEN
      WRITELN(TTY,' DEADLOCK LIMIT EXCEEDED ')
         ELSE IF NETDEADLOCK THEN
            WRITELN(TTY,' OF UNRESOLVABLE DEADLOCK ');
   IF NETTERM THEN WRITELN(TTY,' ALL PROCESSES TERMINATED');
   IF (NETTIME>NETTIMELIMIT) THEN WRITELN(TTY,' TIMELIMIT EXCEEDED');
   IF NETMSGCOUNT>NETMSGLIMIT THEN WRITELN(TTY,' MSG LIMIT EXCEEDED');
   WRITELN(TTY);
   (* RESUME PROCESSES ONE LAST TIME TO PRINT THEIR SUMMARIES ETC. *)
   FOR PROCI:=1 TO HIGHPROC DO BEGIN
      LPS[PROCI].AMPC:=1000;
      RESUME(PROCI);      (* BY CONVENTION, THE LAST CALL TO PROCI *)
      END;
   PRINTSTATISTICS;
   BUFFERSIZE:=BUFFERSIZE+BUFFERSIZE;
   UNTIL (BUFFERSIZE>20);
   WRITELN(TTY,' END PROGRAM DSIM');
END.
```

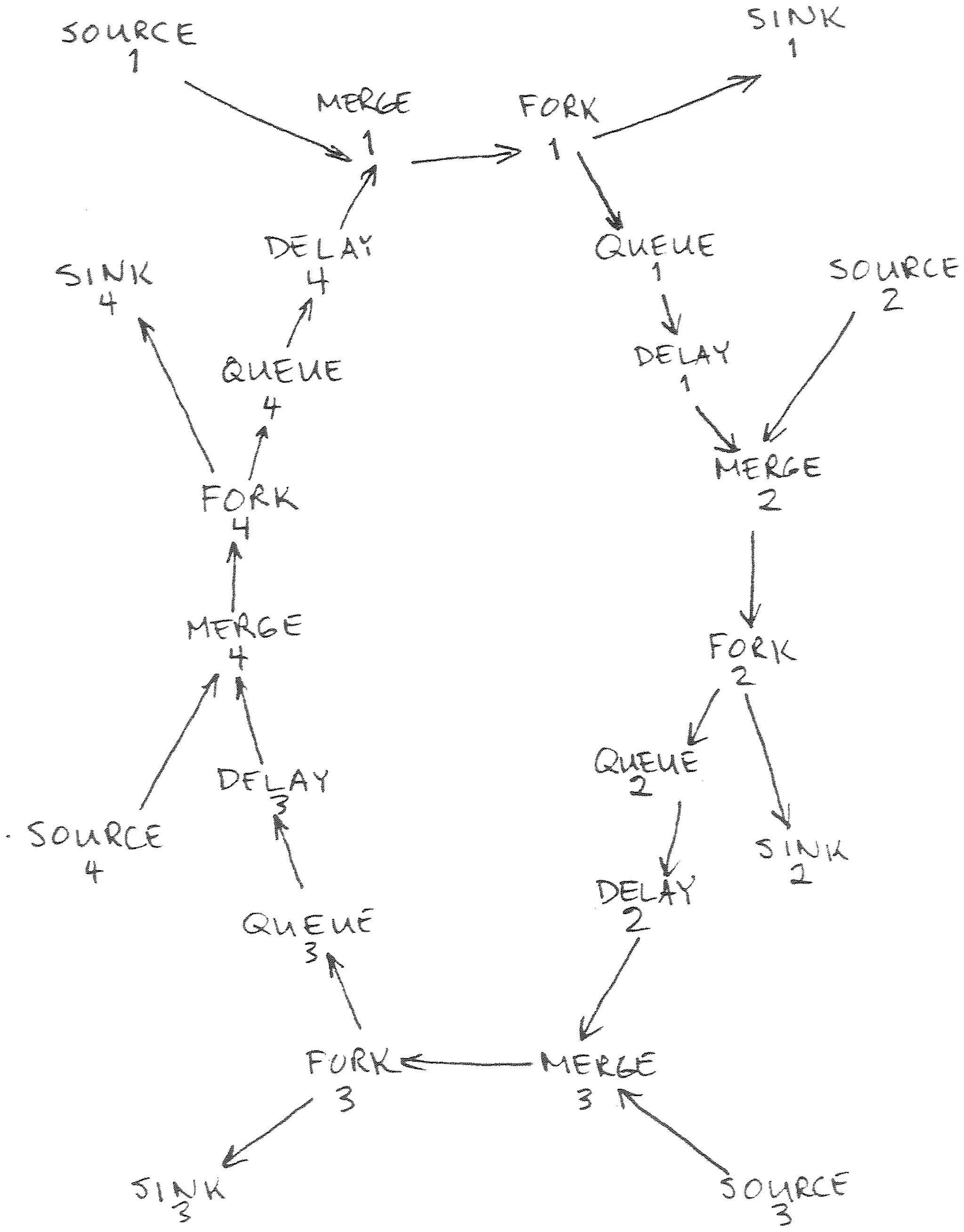
APPENDIX 2

SOME TEST RESULTS

This appendix reports the results of simulation runs on two different networks where the sizes of all the buffers (= queue sizes in processes of type QUEUE20) were successively set to 1, 2, 4, 8, and 16.

The connectivity graph of network 1 is shown in fig A-1. All SOURCE type nodes emitted jobs every 10 time units. The server, or DELAY, nodes had a service time with an exponential distribution, and a mean service time of 7.0 time units. All FORK nodes had a probability of 0.4 of sending a received job out to its respective SINK node.

The relative frequency of job arrivals from SOURCE nodes and departures to SINK nodes was such that the network tended to fill up with jobs over a period of time, and finally encounter a deadlock where all nodes except the SINK nodes were waiting to send out a job. The times at which this occurred depended on the buffer sizes. The following table summarizes the behavior of this network. Elapsed time is the total time from the beginning of the simulation to the final deadlock. MPF represents the "multiprocessing factor", taken as the ratio of the total amount of execution time for all



NETWORK 1

fig A-1

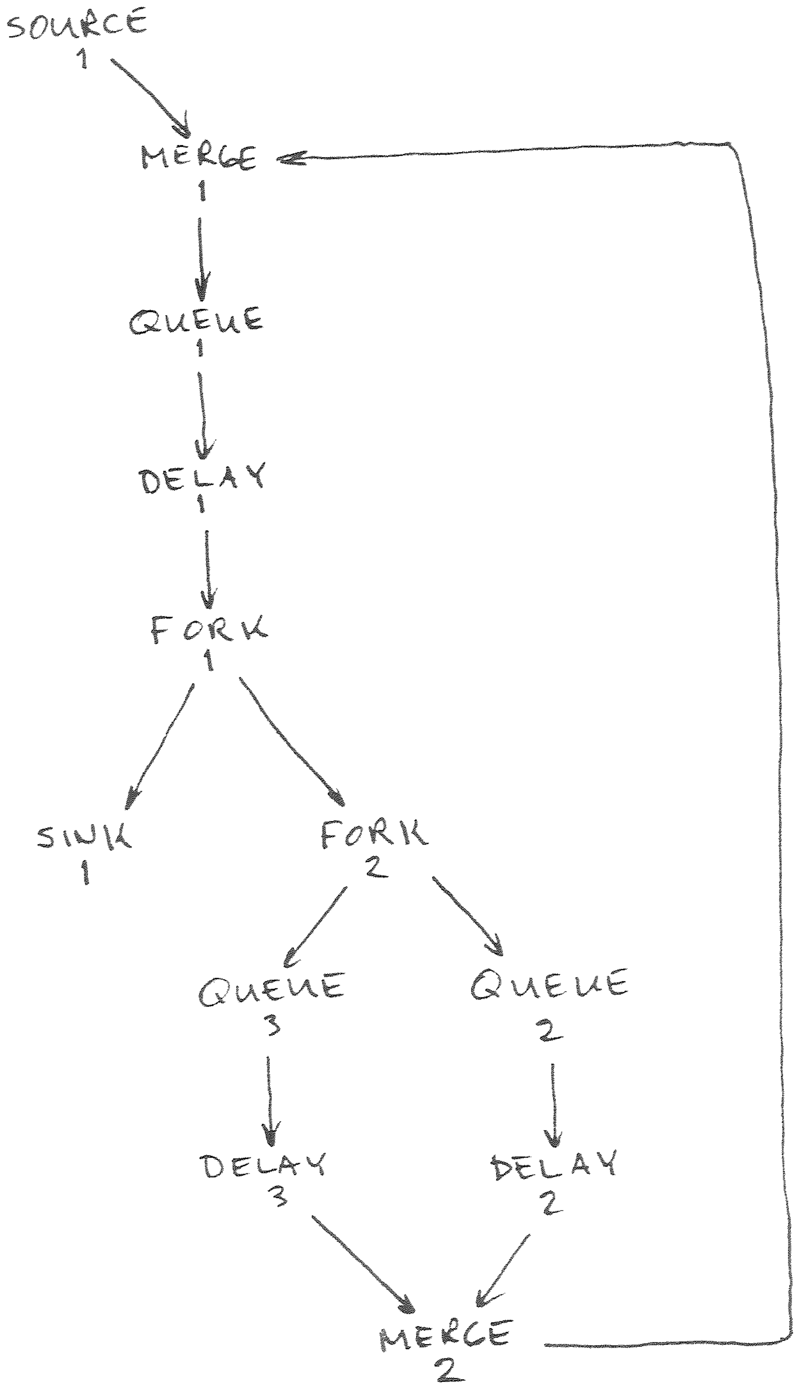
nodes in the network, to the elapsed time for the network.

Buffer Size	Elapsed Time	Execution Time	Deadlock Count	MPF
1	630	2876	2	4.57
2	439	2146	2	4.88
4	891	4794	4	5.38
8	1864	12300	2	6.60
16	1580	10092	2	6.38

In all cases, at least one deadlock was detected and recovered from. With identical job arrivals, service times, and departure times, one would expect the MPF and total execution times to be larger for networks with larger buffer sizes. As the results show, this was not always the case in this run, although the trend is there. This may have been caused in part by short-term fluctuations in the RANDOM function on the DEC-10. This is possible because these results were obtained in a single run of the program, without ever resetting the "seed" of the pseudo-random number generator.

The graph of the second network tested is shown in fig A-2. Here, the SOURCE emitted a job every 20 time units. The node labelled DELAY-1 had a constant service time of 2.0 time units. The other DELAY nodes had exponential service times with mean value = 2.0. Jobs entering FORK 1 were sent to the SINK with probability 0.2. Jobs entering FORK 2 were equally likely to go to QUEUE 2 or QUEUE 3.

Again, the buffer sizes were set to 1, 2, 4, 8, and



NETWORK 2

fig A-2

16, and statistics collected for each size. This network did not fill up with jobs like network 1. Each run continued until the elapsed time was 1001. All of the deadlocks encountered were those arising as a result of the waiting rules for the processes. The following table summarizes the behavior of this network.

Buffer Size	Execution Time	Number of Deadlocks
1	1758	29
2	1724	27
4	1677	24
8	1773	23
16	1702	21

In this network, the larger buffers always resulted in fewer deadlocks, although the net-wide sum of execution time seems uncorrelated.

REFERENCES

[CHA79] Chandy, K.M., J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," IEEE Transactions on Software Engineering (September, 1979), pp. 440-452.

[CHA81] Chandy, K.M., J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," CACM (April, 1981), pp. 198-206.

[HOA78] Hoare, C.A.R., "Communicating Sequential Processes," CACM (August, 1978), pp. 666-677.

[LAM78] Lamport, Leslie, "Time, Clocks, and the Ordering of Events in a Distributed System," CACM (July, 1978), pp. 558-565

[SEE79] Seethalakshmi, M., "A Study and Analysis of Performance of Distributed Simulation," M.S. Report, 1979, Computer Science Department, University of Texas, Austin, Texas.

[DIJ77] Dijkstra, Edsger W., "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," Current Trends in Programming Methodology, Vol I. Software Specification and Design, (Yeh, Raymond T., editor) Prentice Hall, Englewood Cliffs, N.J., 1977, pp. 233-242.

VITA

William Francis Quinlivan III was born in Portsmouth, Virginia, on February 4, 1950, the son of Faye Elizabeth Quinlivan and William Francis Quinlivan, Jr. After attending Central Catholic High School, he graduated from Melbourne High School in 1967. He attended The University of Florida and subsequently served in the U.S. Army. He attended Austin Community College for a year in 1974, and in 1975 he entered The University of Texas. In June, 1978, he received a Bachelor of Science degree in Mathematics, and a Bachelor of Arts in Computer Science. During the following years, he was employed as a Systems Programmer at Tracor, Inc. In June, 1979, he entered the Graduate School of The University of Texas. He is currently employed as a Systems Analyst by Information Research Associates in Austin, Texas.

Permanent address: 609 Orland Blvd.
Austin, Texas. 78745

This report was typed by William F. Quinlivan.