

DESIGN OF A DISTRIBUTED SIMULATION TOOL

by

HARRY L. PHELPS

B.S. CS, Kansas State University, 1979

B.S. CPT, Kansas State University, 1981

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

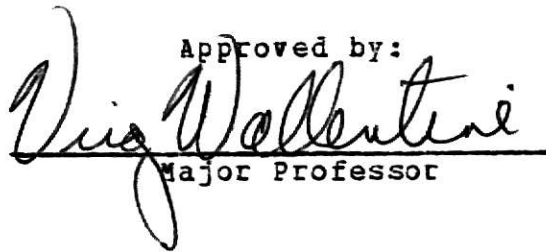
MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

Year: 1981

Approved by:


Major Professor

SPEC
COLL
LD
2668
.R4
1981
P53
C.2

TABLE OF CONTENTS

ILLUSTRATIONS	iii
ACKNOWLEDGMENTS	iv
I. INTRODUCTION	1
A. OVERVIEW	1
B. MOTIVATION	1
C. OVERVIEW OF SIMULATION	3
II. THE ALGORITHM	10
A. CHOICE OF ALGORITHM	10
B. SYNCHRONIZATION	12
C. CONTROLLER PROCESS SOLUTION	19
D. NULL MESSAGE SOLUTION	22
E. ALGORITHM OPERATION	24
III. THE IMPLEMENTATION	30
A. OVERVIEW	30
B. SIMMON	31
C. NADEX	32
D. SYSTEM MODIFICATIONS	38
IV. SUMMARY AND CONCLUSION	42

ILLUSTRATIONS

1.	AN EXAMPLE OF A DIRECTED GRAPH	5
2.	MULTI-NODE CONNECTED GRAPH	5
3.	EXAMPLE OF SHORT ORDER RESTAURANT.	8
4.	COMMUNICATION LINK TIME COMPARISON	12
5.	EVENT TIME CALCULATION	14
6.	MERGING PROCESS EVENT CALCULATION	15
7.	DEADLOCK EXAMPLE	16
8.	UNBOUNDED BUFFER SOLUTION	18
9.	WAITING RING EXAMPLE	19
21.	PROCESS ALGORITHM	10
11.	CONTROLLER ALGORITHM	22
12.	NULL MESSAGE SOLUTION	23
13.	LINK TIME ALGORITHM	29
14.	SIMMON	34
14.	REGISTRATION EXAMPLE	35
15.	HOST MACHINE SEPARATION	36
16.	NADEX COMMUNICATION	38
17.	COMMUNICATION PROCESS	41

ACKNOWLEDGEMENTS

I would like to express my deepest appreciation to the staff of the Computer Science Department, especially those members of my committee: Virgil Wallentine, Beth Unger, Rod Bates and Myron Calhoun. I especially appreciate their support, guidance and gifts of knowledge, for this I will always be grateful, also for the support from my wife, Michal, and my family.

I. Introduction

A. Overview

This report surveys distributed simulation, focusing on the "link time" method of synchronization. Chapter 1 contains a tutorial on distributed simulation, covering terminology and the modeling method. Chapter 2 surveys the "link time" method of synchronization and contains an overview of synchronization problems, and the link time algorithm solution to the problems. Chapter 3 contains a possible implementation of the algorithm using currently available tools at Kansas State University.

B. Motivation

It has been common practice to simulate the behavior of a physical system by the use of another scaled system. The computer far and above has been the most popular device to model such a system because of its great adaptability. Simulation has been one of the most productive and useful applications of the computer. It allows the user to obtain an approximation of the physical systems parameters in a convenient time span that would be impossible to observe in the physical system by conventional methods.

In the past, the simulation of physical processes has largely been by discrete sequential simulation languages such as SIMSCRIPT [6] and GPSS [5], or at times by low level assembler language. The majority of simulation models fit nicely into this realm, but there is a subset that would

perform best in a concurrent environment. These are an interesting class of systems that exhibit a high degree of natural parallelism. Computer systems, communication networks and queuing networks are examples that are inherently parallel, and, if supported by a concurrent language and a multi-processor, potentially would run to completion in far less time than its sequential counterpart.

The cost of computer processing has plummeted over the past two decades. This, coupled with the current availability and growing popularity of the relatively slow but very inexpensive mini and micro computers has brought distributed processing into the lime-light. These systems are an economically feasible and attractive alternative to large main-frame installations. A simulation that would execute on a multi-processing main frame would run just as satisfactorily on a distributed network of smaller machines, given that there is a simulation host and network protocols to support this activity.

A simulation host would provide the simulation primitives and report facilities that are needed to perform model simulations of physical systems. The network protocol is the means by which the respective computers communicate and synchronize their activities.

The performance of the distributed system is dependent upon the network communication protocol and the inherent parallelism of the simulation model. A queuing model that

is run on this type of system should perform significantly better than it would run if it were in a sequential-uniprocessor environment.

C. Overview of Simulation.

Simulations can be grouped into two separate classes: continuous-variable and discrete-event. The continuous simulation systems depict the continuous change of variables within the simulation with respect to time. Physical systems characteristic of differential equations, where time is the independent variable, are of this type. The discrete-event class of simulations change state in discrete time increments and are instantaneous in nature. The following is a common analogy of the two systems: A continuous system can be compared to an analog wrist-watch (having continuously moving hands), whereas a discrete system is the digital counterpart displaying one second intervals. Simulations of the type described earlier (i.e. queuing network, etc.), fall into the discrete-event category.

Time, with respect to computer simulation, most generally is an abstraction or scaled as compared to the real system and the real system's time span. That is, the state of the simulated system at a certain simulated time will correspond to the real system at some corresponding discrete point in time.

A simulation can also be driven in one of two modes of simulated time: event or time-driven. Time driven refers to the simulation clock being updated in fixed increments which define a simulation interval. All of the changes in the state of simulation in the present interval are simulated to that point in time before proceeding to the next clock update. Whereas, in an event-driven system the simulated time increases monotonically (i.e. non-decreasing), but not necessarily in fixed increments. The increments represent times at which the state of the simulation changes, that is, corresponding to events in the system. To achieve the maximum amount of parallelism from the simulated system, the event driven method will be used for the distributed simulation system design presented in this paper.

The modeling of event-driven systems is most easily represented by directed graph structures. A physical process might be represented as a node in the graph and could be thought of as a producer-consumer, source or sink process. The producer-consumer processes have the responsibility of the intermediate nodes in the respective graphs, whereas a source is a process with no input links and a sink is a process with no output links.

```
*****      *****      *****      *****
*          *          *          *          *          *
* Source *----->* Process i *----->* Process i+1 *----->* Sink *
*          *          *          *          *          *
*****      *****      *****      *****

Source                Consumer/Producer        Sink
process              process                    process
```

Figure 1 illustrates a Source process connected to a consumer/producer process and then it, in turn, is connected to another consumer/producer. The last process is a sink. the inter-process communication would proceed in the direction indicated by the arrows.

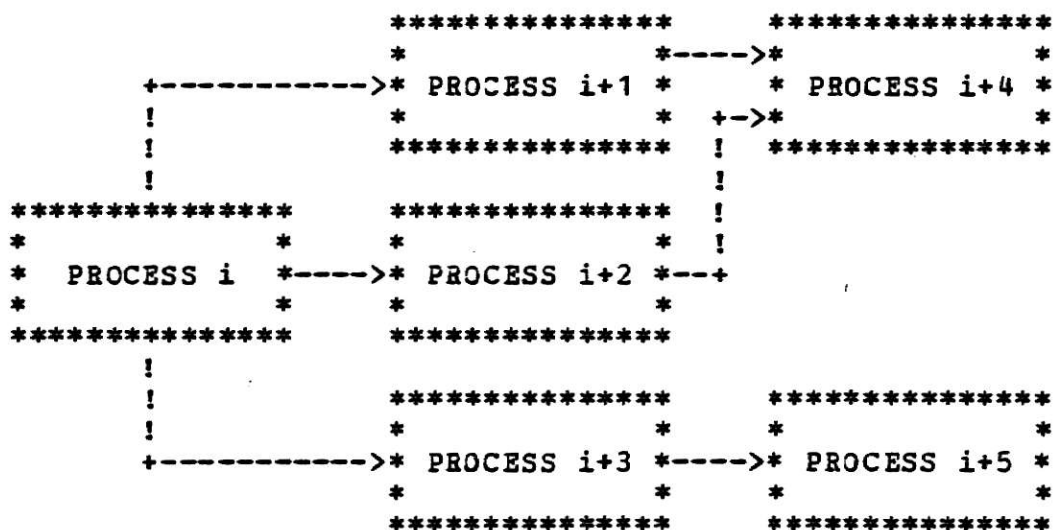


Figure No. 2

By definition, the event times at each node must not decrease in simulated time sequence. This allows a node to merge the events arriving at that node via its input links into the correct time sequence.

In the area of multi-processor systems there is yet another classification of simulation methodology that must be applied to the design of this system. This classification is that of tight or loose-coupling. These variants refer to the maintenance of nodal simulation time (or process clock values). If each nodal process is allowed to monitor its time independent of the nodes in the graph it communicates with, then it is called loosely-coupled. But if all nodes in the graph maintain the same clock time, then it is called tightly-coupled.

The loosely coupled system allows the most freedom to each node. Any node or process whose sequence of states does not depend upon its ancestors is allowed to proceed to the next event at that node whenever possible. This concept blends nicely with the desired performance of the envisioned simulation system. By using the loosely-coupled event-driven characteristic for the design of this system, each process achieves the greatest amount of autonomy by allowing each process to step its simulation time forward to the time of the earliest event at that node. To achieve this system characteristic requires the execution of an algorithm in each component node to determine, based on the information

received from its ancestors which influence its state changes, whether or not it can step its simulation time forward.

Micro and mini computers, when configured into a directed graph configuration, form a network called a loosely-coupled distributed system. Because these machines share no memory, they are forced to communicate over cable or telephone link communication systems. This is the type of computer network that will most likely be available to the user of a distributed network, and most other network topologies can be configured to represent a directed graph if necessary.

How can a network of micros be configured to represent some physical system that is of interest to the user, and at the same time achieve the greatest amount of parallelism from the individual machines. The machines must be kept executing user code at any particular point in time. The basis of this system is built around the concurrent structuring concept. To get a better idea of what is involved in this area, it is now time to enlist some help from one of the patriarchs of concurrent programming, Per Brinch Hansen. He states in reference [3] that he invents a program structure "...by drawing pictures of it from different viewpoints over and over again until a simple and convincing pattern emerges". The picture that results takes the form of directed graphs normally representing data flow

or access. This is what must be done in the conversion of a physically concurrent system to a logically concurrent system for computer simulation. Those actions that are characteristic of a particular physical process should be included in that logical process and those pertinent to another physical process should be grouped in yet another logical process, within the system configuration. A model of a short order restaurant might be represented by:

EXAMPLE OF SHORT ORDER RESTAURANT

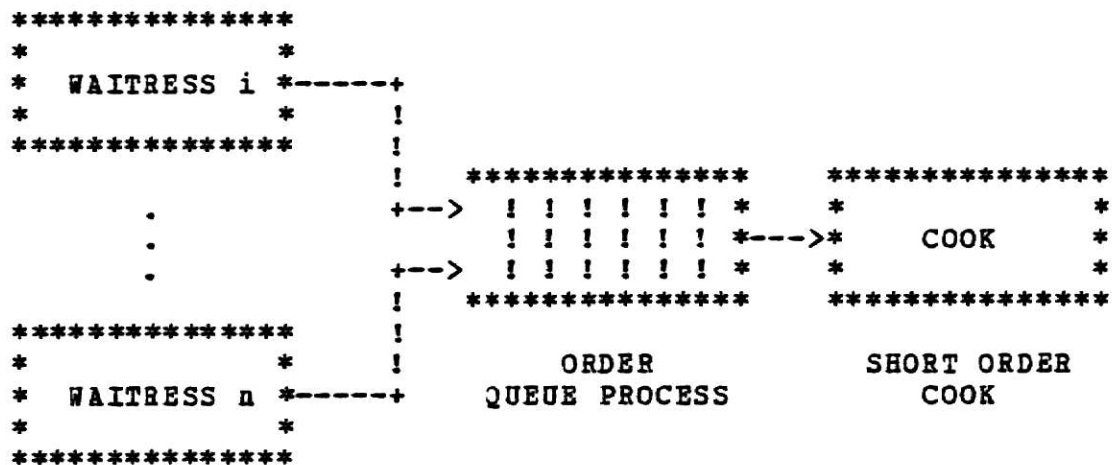


Figure No. 3

This scheme can be used on any physical system that can be partitioned into a set of parallel processes that communicate with one another exclusively via messages. Any interaction between physical processes can be modeled as a message sent between the respective processes. The physical system that is to be simulated will be represented by a directed graph, and there will be an arc connecting the adjacent nodes if and only if the respective processes communicate with each other. A message is sent by a producer process to a consumer process at any point in real time in the physical system to represent an event in the physical system. This action can be modeled by a message tuple (TIME,MESSAGE) from the producer process to the consumer process in the simulated system at that point in simulated time. The system is asynchronous. The key is the encoding of time as part of the inter-process message. By the encoding of time in the message, synchronization of the system's processes can be achieved without the use of a global clock.

Given a network of processors, the simulation can now be decomposed into its components and distributed over the network. An ideal implementation of this system would be to allocate one process per processor. In this way the greatest amount of parallelism would be achieved, for no time sharing would be needed in this system.

II. The Algorithm

A. Choice of Algorithm

Earlier discussions have centered around the need and description of a distributed computer network along with the conceptual modeling of physical systems. Now, an algorithm must be developed to drive each component process within the network. This algorithm was proposed by Peacock, Wong and Manning in reference [10] and in references [9,12] by Chandy and Misra.

This algorithm has been explicitly designed for the loosely-coupled event-driven network of computers and should provide the parallelism necessary to model concurrent systems efficiently. As mentioned in the introduction the loosely coupled method of simulation will allow a producer process as a member of a directed graph, to update its clock to the time of the earliest event at that server. This property of the envisioned system could provide reasonably good performance from an inherently parallel simulation model. Given that a node commands this amount of autonomy, how does it determine the earliest event time and what synchronization problems arise in these graphic configurations? These questions need to be answered for a complete understanding of the network operation and for some insight to the performance characteristics of the distributed network.

As mentioned earlier the only means a process has to

communicate to its adjacent process is via a message (TIME, MESSAGE). This communication can be based upon any protocol, but to save buffer space a simple protocol, designed by Hoare, will be used for the description of the algorithm. A message is transmitted from the producer to the consumer if and only if the producer is waiting to send a message to the consumer and the consumer is waiting to receive a message from the producer.

Two restrictions must be placed on the system to ensure the proper process synchronization and that the simulated system actually represents the true physical system. These are:

- i) Event times across the link are monotonic--ie. non-decreasing. (an event-driven attribute).
- ii) The output of a producer at any given time depends only on the messages it has received before that time and its own logic. (All physical systems exhibit this characteristic.)

These restrictions imply that at any given time a producer may decide to send a message to a consumer. The decision to send a message and the content of the message are determined by the producer from the messages it has received (its message history). The behavior of a producer or consumer at any point in its simulated time cannot be influenced by messages received after it has updated its clock beyond that point in time. This implies that if a producer sends a message to a consumer at some point in its local simulation time then all messages have been simulated

up to that point in time. The restriction of non-decreasing event times is a characteristic of naturally occurring systems and cannot be violated. This is also reflected within the simulated system and will result in erroneous results if simulated time were allowed to decrease with respect to physical time in the system.

B. Synchronization Problems

In a system in which processes communicate exclusively with messages, there are bound to be synchronization problems. Figure (4) depicts such a problem:

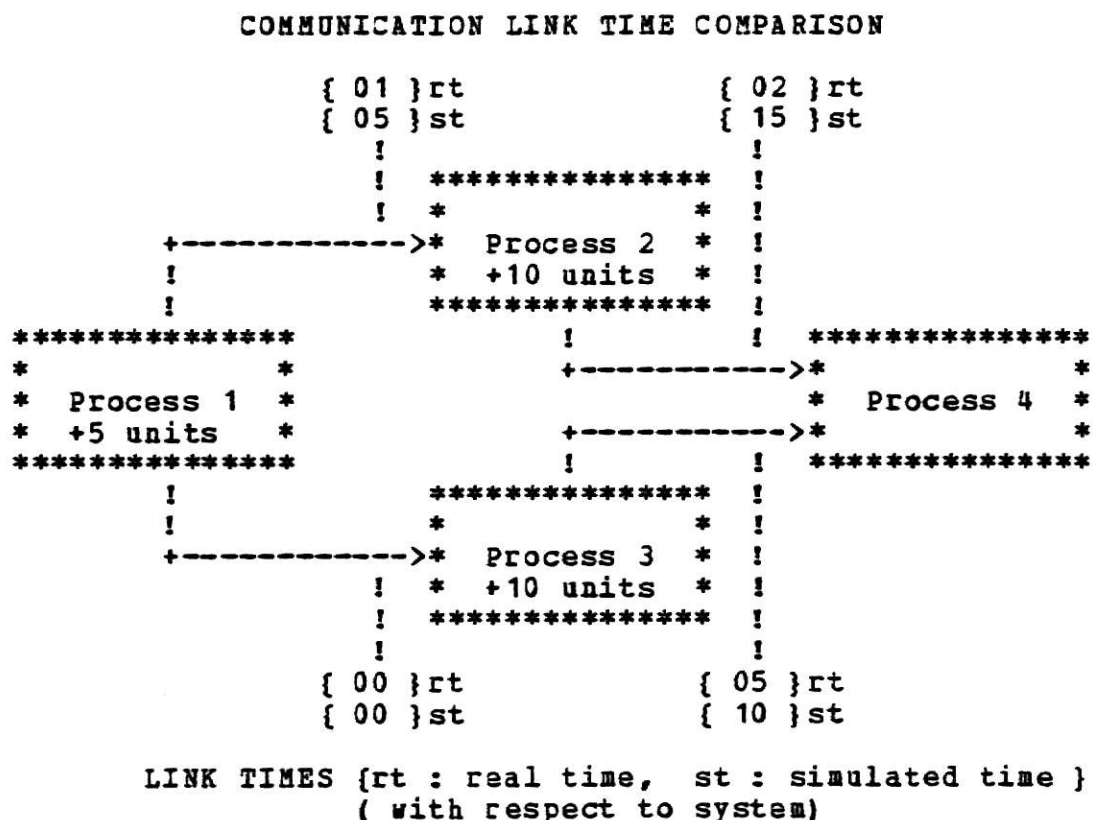


Figure No. 4

Figure 4 illustrates that a physical message could arrive at process 4 by the upper route in less time than the lower route, but after it in simulated time.

A message sent from node 1 to node 3 and then on to node 4 could be proceeded in real time by a message sent the 1-2-3 route even though it was generated later than the first message due to slight variations in link speeds. This is due to the instantaneous time updates characteristic of the simulation system. That is, in real time the two messages will be sent practically simultaneously so that in all practicality the second message generated could arrive ahead of the first logical message.

To resolve this problem a convention must be accepted. A process must not update its local event clock until it has received the next time update from all its producer processes. To implement this convention the consumer and producer processes must maintain a link time for each arc or link connecting the respective nodes of the graph. This link time is to that point in simulated time which the two processes have simulated the inter-connecting arc.

A process can calculate its lookahead or next event by selecting the minimum arrival time as given it by its producers as shown in Figures (5) and (6).

EVENT TIME CALCULATION

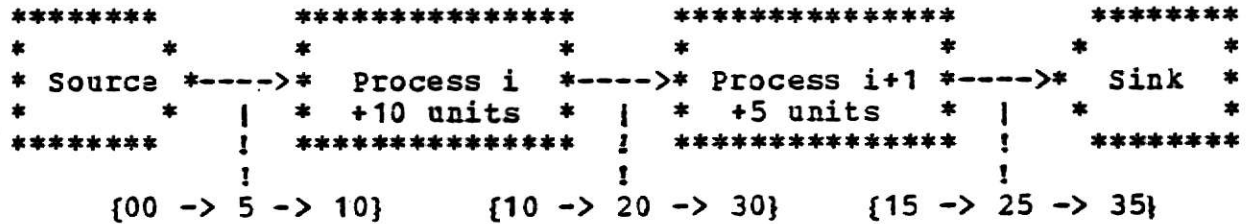


Figure No. 5

This pipeline model illustrates the calculation of the lookahead of the next event at the respective node. Process i adds 10 units to the time it has received from the source. It then transmits the message to the next process i+1. It then updates it's local clock to that value. The same procedure is followed at process i+1, and then finally by the sink process.

MERGING PROCESS EVENT TIME CALCULATION

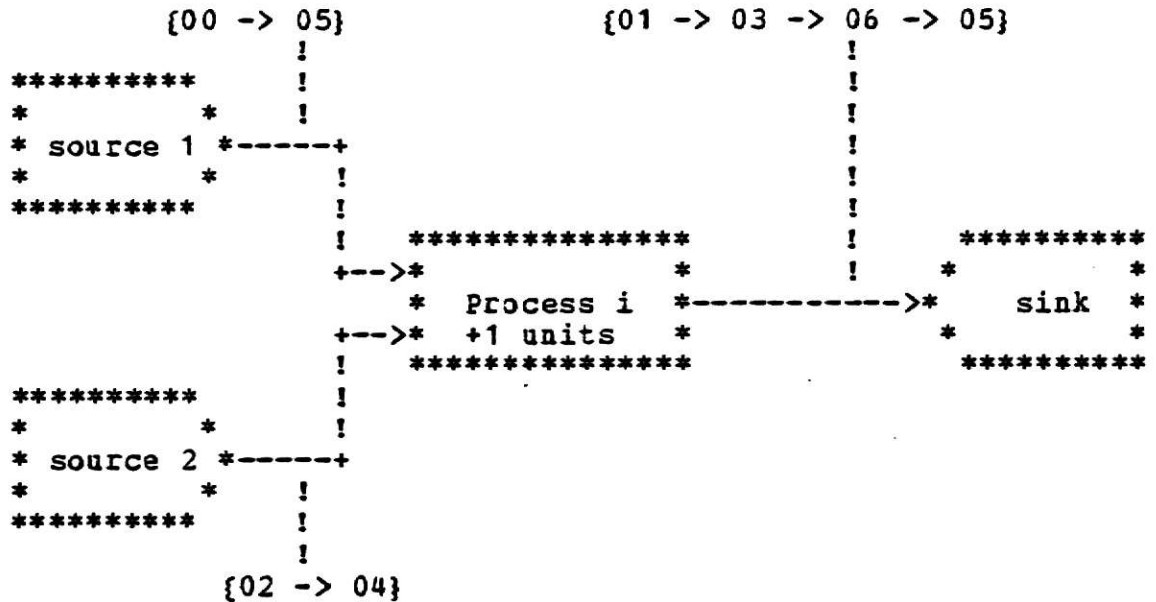


Figure No. 6

The next event at process i would be the minimum arrival time as given it by its producers.

But if there is no message on an input link, as depicted in Figure 7, the consumer is undecided as what to do and no further computing can proceed at that node. The system is deadlocked until a message is sent to the process via the lower link to resolve the stalemate. A lower bound of all arrival times at that node is needed to insure the next event is the true next event.

DEADLOCK EXAMPLE

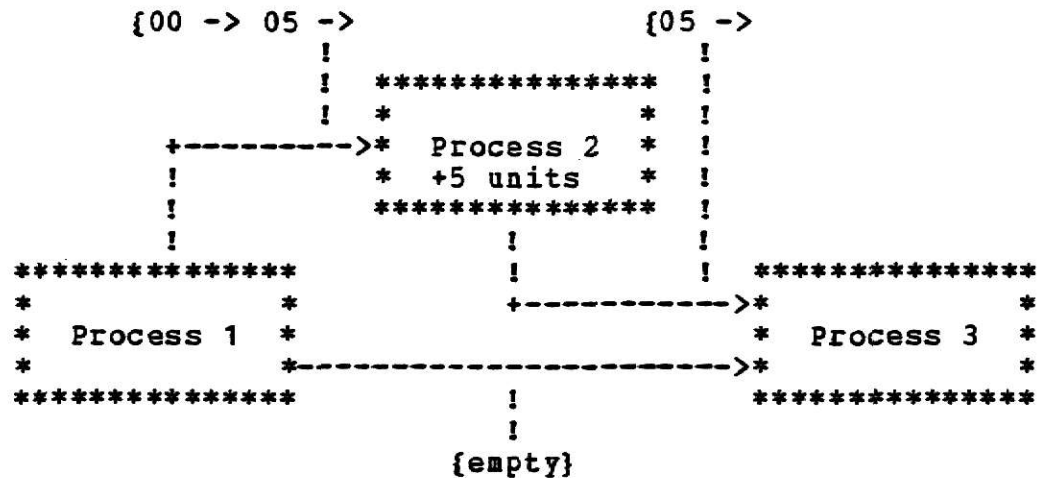


Figure No. 7

GIVEN:

Process 1 sends messages @ 00, 05, 10 to process 2
 Process 1 sends messages @ 20, 40, 60 to process 3

 Process 2 takes 5 units to perform simulated task.

SEQUENCE OF EVENTS:

Process 1 send a message (00,M1) to Process 2
 Process 2 send a message (05,M1) to Process 3
 Process 3 now waits on Process 1 (empty link)
 Process 1 send a message (05,M2) to Process 2
 Process 2 waits to output (10,M2) to process 3
 Process 1 waits to output (10,M3) to Process 2
 Deadlock.

There are three solutions proposed for the synchronization problems given. The first solution will only partially remedy the problem. If there is unbounded storage at the process node for message storage then a process can maintain a lower bound on the event time communicated to the process by its producer process, and hope in time that an input in the empty link will eventually arrive as shown in figure (8). This will allow the process to update its clock to that event. Using this solution on the surface seems to have solved the problem. The processes must eventually communicate or the link would not have been created. However, there are some process configurations which this solution will not free from this stalemate. In the configuration shown in Figure (9) each of the outer nodes is waiting on the adjacent node to communicate. This type of problem is exemplified by the classic dining philosophers situation.

{00 -> 05 -> 10} {05 -> 10 -> 15}



(18)

WAITING RING EXAMPLE

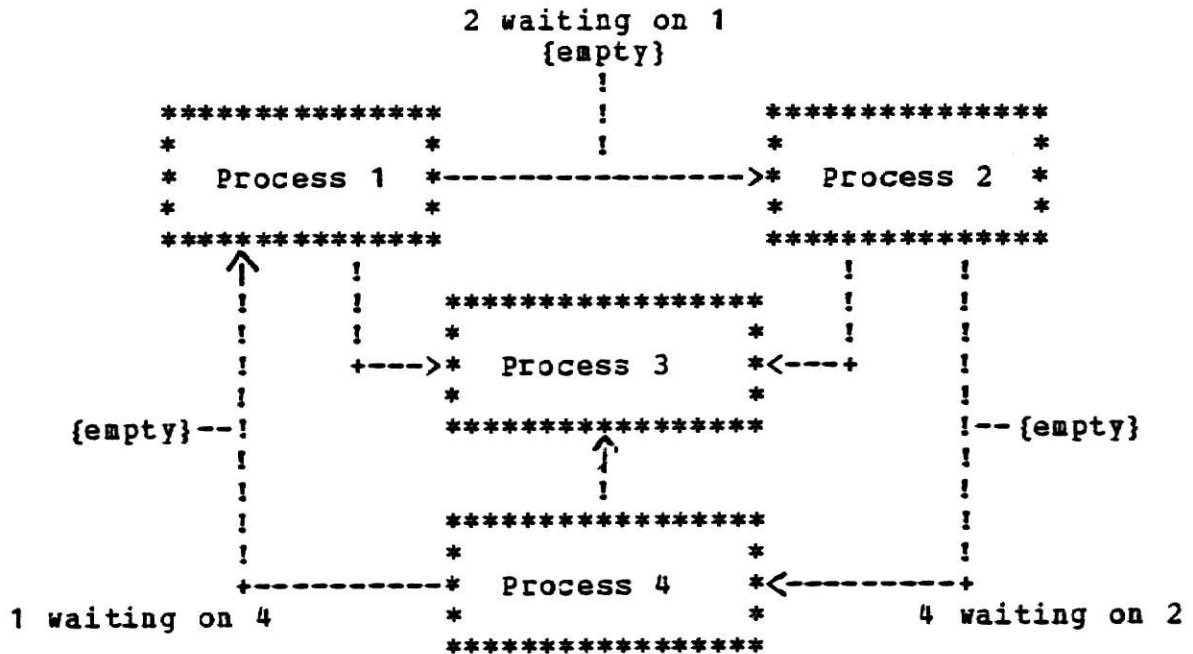


Figure No. 9

In this configuration, if each of the outer links becomes empty then deadlock will occur.

C. Controller Process Solution

The second solution is to allow the network to deadlock and then initiate a computation whereby the various processes can update their local clock values. This would involve the creation of a special process whose sole purpose

is to synchronize these actions and resolve deadlock.

The algorithm performs in the following fashion using the following sequence of computations:

- i) Parallel phase: Run simulation until system deadlocks.
- ii) Phase interface: Initiate a computation whereby the various processes can advance their local clocks.

When the network deadlocks the interface phase is initiated. In a sequential simulation of the event-driven type, each process logs the time of its next output into an event-list, assuming it receives no further inputs. This list is maintained by the simulation host. The event-list is scanned for the smallest logged time and this value then becomes the next event within the system being simulated.

The same method for resolving the deadlock can be initiated with the concurrent simulation. All processes within the simulation are waiting to either input or output a message tuple (TIME,MESSAGE). This tuple corresponds to the next possible event at that respective node being serviced by its server processes. These event times correspond to an event list within the sequential simulation. The controller process detects deadlock and requests communication of the clock values from each of the processes within the simulation. The controller then uses these times to calculate a lower bound on the next event time to be transmitted to each of its consumer processes involved. Once this lower bound has been determined the

corresponding event time and message tuple is updated at this node, and the corresponding tuple transmitted.

This sequence of events is repeated until all processes have been restarted. It can be shown that there is at least one restartable node within the deadlocked network (ref. [12]). The algorithm that each process must execute to implement this deadlock solution is given in Figure (10), and the algorithm for the controller in (11).

PROCESS ALGORITHM

Algorithm for each Process

```
initially: clock value is 0 for every line
           (hence Process clock value is 0 for every Process);
           "Define this Process to be resumable if it is
           waiting to output along some line;"

LOOP:
    Communicate with controller:
        "this phase is entered initially and upon completion
        of minimum-event computation"
        send a signal to controller stating whether or not
        this Process is resumable;
    Simulate:
        using the algorithm of Figure [12]
    Compute Minimum of Process event times:
        "this phase is entered upon detection of deadlock.
        Controller sends a signal to each Process to enter
        this phase."
    event time := minimum accross input links from ancestors
End-loop

End-algorithm-for-Process.
```

Figure No. 10

Algorithm for the Controller

Loop:

receive signals from all Processes as to whether or not
they are resumable;
receive signals denoting deadlock;
send signals to all Process's to initiate minimum-event
computation

End-loop

End-algorithm-for-controller.

Figure No. 11

D. Null Message Solution

The third solution involves the creation of NULL messages. At each fork of the directed graph null messages are sent to the consumer processes. In this case null means carrying no state change information for the next event. The time component of the message is the next event time for which the consumer process can calculate its lookahead. This allows the consumer process receiving the null message to update its clock to a new lookahead value, and resolve the deadlock, thus not allowing any simulated time update to go unnoticed by a merging process.

NULL MESSAGE SOLUTION

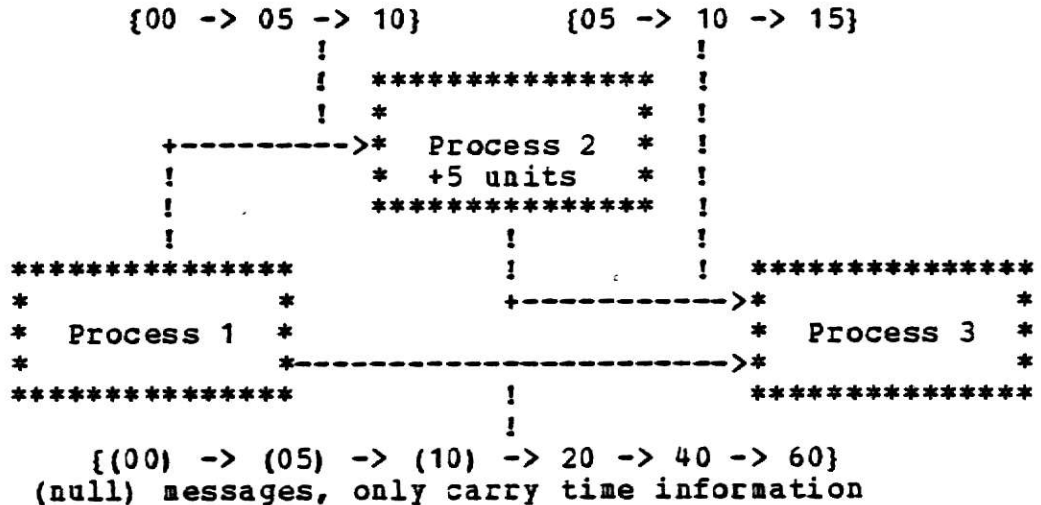


Figure No. 12

Deadlock is avoided by process 1 sending the time stamped "NULL" messages over the link to process 3.

As can be seen, deadlock is avoided "from the logic of each of the individual processes even though each process's logic is independent of the overall structure of the network" [9]. If the physical system is prone to deadlock then the modeled system will also deadlock in a similar fashion. Time and null messages will continue to flow through the network, just as the physical system. If the physical system deadlocks, time continues to advance but all activity corresponding to events within the system have

stopped. The simulated system should also behave in this manner.

E. The Time Link Algorithm

After the initialization of the simulated system, each process within the connected graph repeats the following sequence of events until the simulation terminates:

- 1). (selection) -> a set of lines are selected for which parallel I/O will be initiated for the cycle.
- 2). (computation) -> for every line selected above, determine the next (TIME, MESSAGE) tuple to be transmitted along that line.
- 3). (I/O operation) -> carry out the parallel I/O operation for the set of lines selected in step (1).

1) Selection

The selection of the set of input and output lines is based upon the clock value of the process, in that, this is the clock value of that node and this value of time for any given consumer or producer is defined as the maximum time satisfying this requirement:

All subsequent messages (TIME, MESSAGE) sent or received by a process i must have a value of time greater than the nodal clock value.

Thus the set of possible communication links is that set of all input lines with a clock value equal to the value of the clock at that process unioned with those output lines with clock values equal to or greater than the lookahead of

that process. The clock value of a line is that value to which both processes on each end of the arc have simulated it. The value of the logical process clock is that value to which it has been simulated. Before updating its local clock the process must communicate on all lines calculated in the selection phase. These link time updates must be communicated to the consumer process so that their next event can be calculated. All output lines with values of the line clock that are equal to the value of the lookahead (next event) are not included in the set of available communication lines.

2). The Computation Phase Algorithm

From the history of an arc a determination of the node's state can be assessed and the proper (TIME,MESSAGE) tuple sent to the consumer process. The history refers to the sequence of all tuples (TIME, MESSAGE) corresponding to messages sent at or before the present value of simulated time, that is, it is a complete history of the messages transmitted along the respective arc at or before that point in physical time.

There are two cases for which a tuple must be sent over the arc, These are:

- Case 1 - Where there is a message to be transmitted across the arc in that time period, and for this case the value of the tuple would be (TIME, MESSAGE).

Case 2 - Where there is no message to be transmitted to the next node, implying a state change, but still a "NULL" must be sent, to allow the receiving process to update its respective local clock.

The value of the time output on the arc is greater than the previous value of the arc and lesser than or equal to the value of the next event at that node.

$$\text{Time(arc)} < \text{Time(message)} \leq \text{Time(next event)}$$

The output operation will increase the old value of the arc to the value of the next event computed in this phase after the transmitting of the (TIME, MESSAGE) value. If the tuple contained a NULL message then the local clock is updated to the value of the lookahead after transmission. Case 1 calculates all the non-null message values whereas Case 2 computes all null message values.

3) I/O Operation

I/O operation involves waiting in parallel to input and output along the lines selected in phase 1 along with the process of updating the link clock value and the value of the message for every selected I/O line. If the tuple (TIME, MESSAGE) is received or sent on any of the selected lines, as a result of the selection process, the value of the last message and time are immediately updated within the logical process to the transmitted tuples value.

Initialization

The local variables needed for each nodal process are:

T_LST_REC, M_LST_REC -> last tuple received on a certain line at the beginning of the selection step for each input line initially set to ZERO, the line has been simulated up to time ZERO. The message is set to NULL.

T_LST_TRNS, M_LST_TRNS -> maintained for each output line to represent the last tuple transmitted on that line at the beginning of the selection step, initially set to (0, Null).

T_NXT_TRNS, M_NXT_TRNS -> denote the next tuple to be transmitted in every output line. This value is computed.

T_L_CLK -> the clock value of the logical process. This is initially set to 0.

Termination

For system termination, it is required that the simulation continue to the value TIME_TERM (clock value at termination) as long as the clock value of the process has a value of less than this time the process must execute the three steps given earlier. The process' line clock should have a value equal to the TIME_TERM at termination. But, there is a chance for the clock to have a value of greater the TIME_TERM, in this case the value should be set equal to the value of TIME_TERM, because no events at his node will occur between TIME_TERM and the newly calculated value. This line should not be used for further I/O.

Performance

The performance of the system is directly affected by the efficiency of the communication system, the interaction of the individual processes and the synchronization of the process. The communication of the process involved the network as a whole and the protocol used may well vary with performance in the configuration and also by the communication process itself.

Any loop within the system will reduce the performance of the system, for the processes within the loop are required to maintain the same local clock value. The synchronization of the process, at least in using algorithm, is expensive, in that the use of null messages cause problems in the production and volume of messages transversing the network. Every fork in the graph doubles the number of messages handled by the network as a whole.

A summary of the algorithm is presented in Figure (13) and a full proof of correctness of the simulation can be found in [9].

LINK TIME ALGORITHM

PROCESS i = PROCESS

"Declaration and Initialization of Variables"

T_LST_REC := 0; " for all "
M_LST_REC := 'NULL'; "inputs"

T_LST_TRNS := 0; " for all "
M_LST_TRNS := 'NULL'; "outputs"

T_L_CLK := 0;

WHILE T_L_CLK < TIME_TERM DO;

 "Select set of input/output lines"

 I/O_SET := {INPUTS | T_LST_REC = T_L_CLK}
 union {OUTPUTS | T_LST_TRNS = T_L_CLK <= LOOKAHEAD};

 "Computation of next event, next message tuple"

 FOR OUTPUTS IN I/O_SET DO;
 IF CASE1 THEN TUPLE(LINE) := (TIME, MESSAGE);
 ELSE TUPLE(LINE) := (TIME, "NULL");

 IF TIME > T_TERM THEN TUPLE(LINE) := (T_TERM, "NULL");
 "Input and Output Operation"

 FOR OUTPUTS IN I/O_SET DO;
 WRITE(TUPLE(LINE));
 T_LST_TRNS, M_LST_TRNS := TUPLE(LINE);
 END; "DO"

 FOR INPUTS IN I/O_SET DO;
 READ(TUPLE(LINE));
 T_LST_REC, M_LST_REC := TUPLE(LINE);
 END; "DO"

 "Compute local process time"

 T_L_CLK := MIN(T_LST_REC, T_LST_TRNS)

END; "WHILE"

END; "Process"

Figure No. 13

III. The Implementation

A. Overview

The implementation of the proposed algorithm involves these two aspects: 1) a simulation host, and 2) a network protocol or operating system which will support this type of process communication. Most simulation languages do not provide the desired structuring ability needed to model nodal processes as a connected graph, nor do they provide the synchronization primitives needed for this environment.

It goes without saying that a concurrent program probably would not perform as well as its sequential counterpart if only one processor were allocated to the program as a whole. The increased speed is achieved when the program's processes are allowed to execute concurrently using more than one processor to distribute the workload. A concurrent programming language simply forces the programmer to think in a concurrent fashion along with providing the concurrent primitives and environment to exercise this freedom. Concurrent Pascal is a language that was designed and implemented by Per Brinch Hansen and does provide the concepts of classes, monitors and processes needed to implement a system based on a concurrent environment. Concurrent Pascal was created specifically to implement what could be termed as "event-driven" systems such as operating systems and real time control systems. It would naturally follow that it would be a proper choice for this system

design.

B. SIMMON, Simulation Host

SIMMON is a simulation monitor that has been developed at Kansas State University and is based on concurrent Pascal. It provides the simulation primitive and synchronization needed to construct and implement directed graph simulations. SIMMON is based on SOL, a simulation language that describes the features that have been implemented within this monitor. SIMMON, as it stands, is a tightly-coupled event-driven simulation monitor and executes only within a uni-processor environment. It will need to be modified to support both network communication and loose-coupling for this system design. The goal of the envisioned system is to obtain the greatest amount of inherent parallelism from the modeling processes. To be able to achieve this goal the system must exhibit the loosely-coupled event-driven system characteristic.

The SIMMON monitor allows the user to construct scenarios or concurrent processes that represent the nodes of the directed graph figure (2). Each node contains the information and code to determine its actions or responsibilities with respect to the system. These scenarios are actually sequential Pascal programs that are synchronized via the monitor itself and contain calls to this monitor at points of shared resource access and event logging. This allows the monitor to synchronize the use of

the shared resources along with collecting statistics on the performance of the simulated system. Some of the primitives the SIMMON monitor offers are:

- i) "SEIZE" and "RELEASE" a facility
- ii) "WAIT-TIME", "WAIT-RAND"
- iii) "INCREMENT_COUNT"
- iv) "SIGNAL" and "WAIT" for an event

C. Nadex Operating System

SIMMON will be interfaced to Network Adaptable Executive (NADEX) a network operating system which was developed and implemented at Kansas State University. This facility provides the communication protocol and lays the groundwork for the structuring of distributed communication between host machines.

NADEX is a message based multi-user network operating system, and is also written in C-pascal. Nadex provides the facilities for constructing software configurations. These configurations can be thought of as being made up of componets. Each componet may consist of combinations of other componets or subcomponets. A node, which is implemented by a process in the Core OS is the most primitive componet and can perform one of two functions. It may provide access to other system resources or can perform as a Concurrent or Sequential Pascal program. Communication

between components is achieved by naming ports [11] of each node that are to be connected and then "READ"ing and "WRITE"ing on these buffered ports. This connection is established by a DTS-DATA TRANSMISSION STREAM. NADEX allows the user to construct these configurations at interpretation time. These configurations are dynamic in nature and can contain cycles.

Figure (15) contains a diagram of a three machine model of school registration. It is a fifteen process implementation of the distributed system going from the physical system to the logical model and then to the physical computer system organization.

```

! #####
! # 2 #
! #####

*****
* modeling *
* Process i *-----!----->*
* (scenario 1)***
*****
*
* -----!----->*
* scenario 2 ***
*****
* . . .
*
* -----!----->*
*
* -----!----->*
* scenario n *
*****

#####
# 1 #
#####

-----USER PROGRAMS-----

#####
# 3 #
#####
* SIMMON MONITOR
*
* 1) SOL LIKE ENTRY PROCEDURES
* 2) SIMULATION PRIMITIVES
* 3) SYNCHRONIZATION FACILITIES
* 4) REPORT FACILITIES
*
*
*****

#####
# 4 #
#####

1: Logical modeling sequential
   programs.

2: Upon program execution
   sequential programs are
   loaded as concurrent programs.

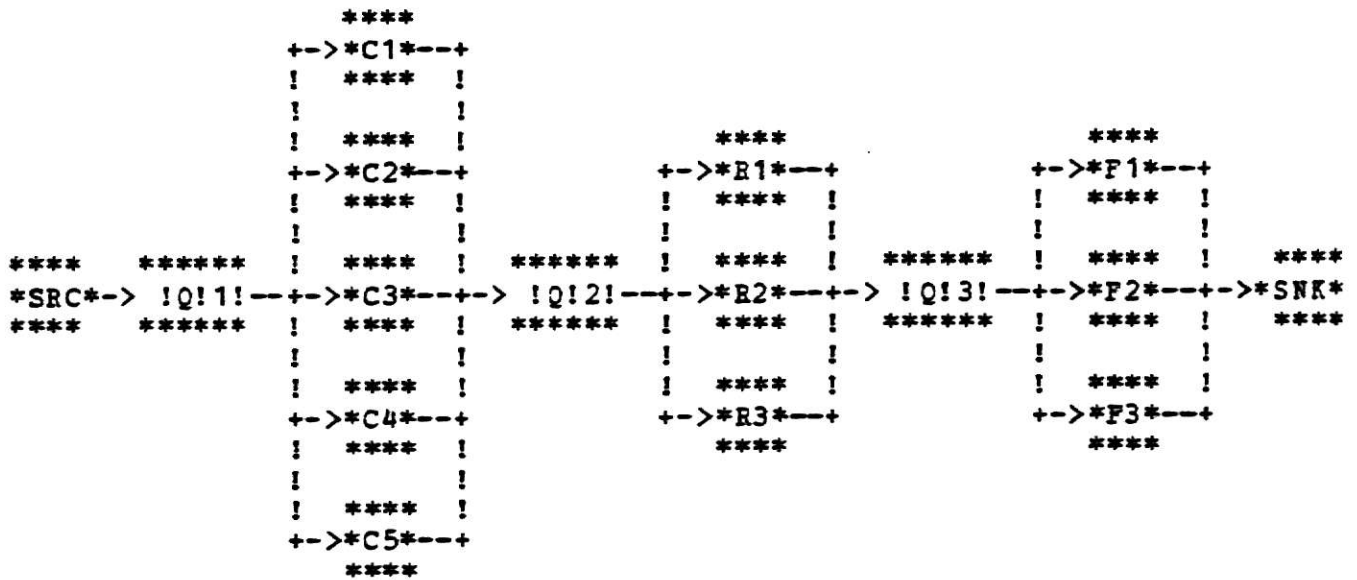
3: The modeling concurrent
   processes access monitor
   simulations primitives.

4: Static concurrent processes
   provided concurrency in monitor
   activities, ie. communication, etc.

#####
* STATIC *
*
* Process 1 ***
*****
*
* -----!
* Process 2 ***
*****
* . . .
*
* -----!
* Process n *
*****

```

Figure No. 14



Line No. 1	Counselors	Line No. 2	Registration Tables	Line No. 3	Fee Tables
---------------	------------	---------------	------------------------	---------------	---------------

This model is of school registration where the normal procedure is to arrive at the gymnasium and form a line to wait to visit with a school counselor and determine your schedule. After finishing with your counselor you enter another line to register with your respective college. After registering the student again forms a line to pay fees. Upon completion of that station the student exits via the gym door.

This model contains:

- 5 : counselors
- 3 : registration tables
- 3 : fee payment tables
- 3 : lines or queues
- 2 : doors (source & sink)

Figure No. 15

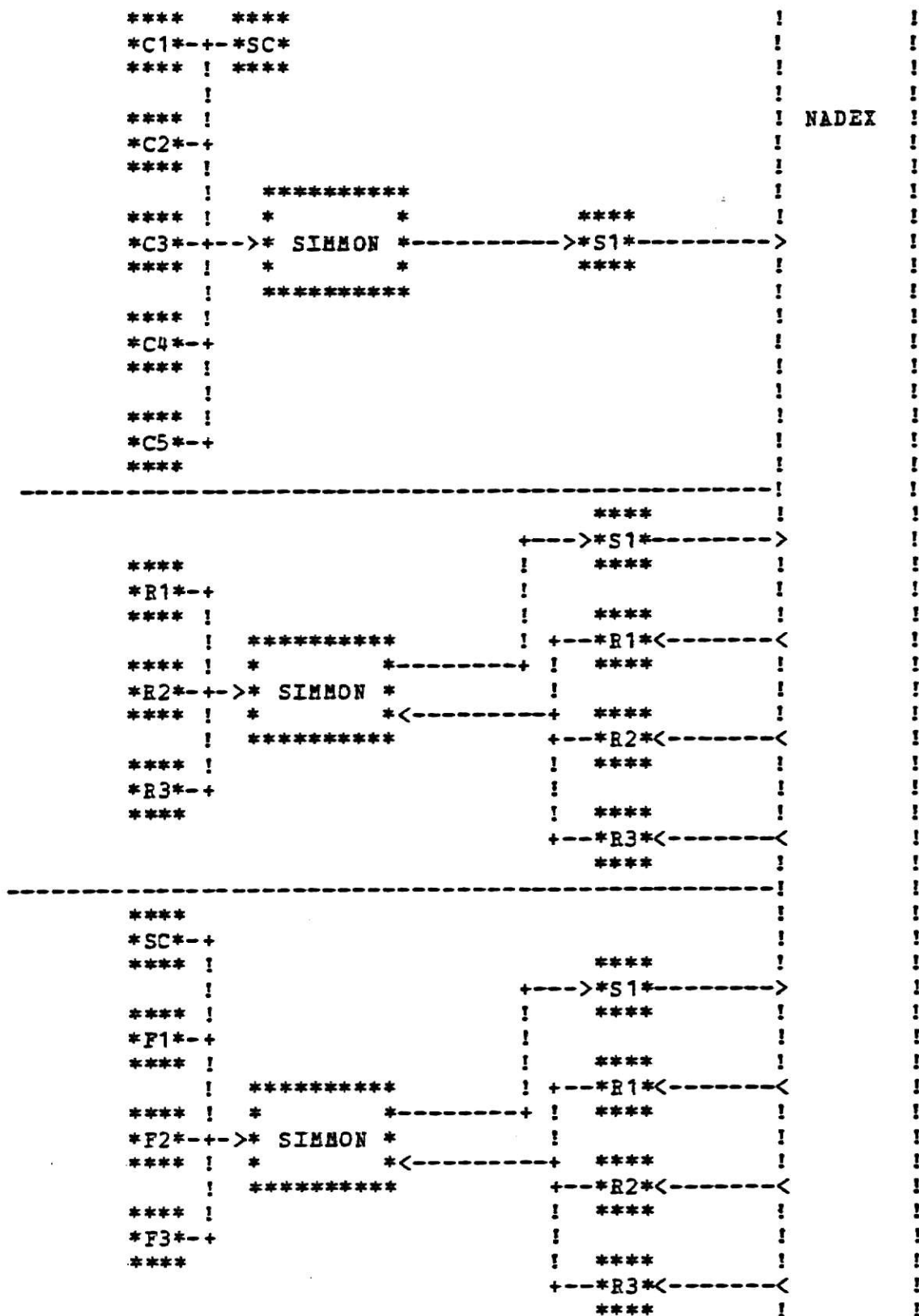


Figure No. 17

Going from the logical model to the computer model state involves the breaking down of the system to combine those processes that interact most often and placing those upon a single node of the computer. If more computers are available a further breakdown can be made until only one scenario belongs to a single host computer.

Each of the host machines execute a resident SIMMON monitor by which the local processes synchronize. The monitor coordinates and collects the data on the local processes and provides calls to the NADEX operating system to communicate with a SIMMON resident on another host machine. In this way resources shared between machines and synchronization primitives needed to coordinate processes which communicate with distributed processes on machines across the network. The monitor's activities will not interfere with the algorithm functions, nor will it substantially degrade the network performance. It is needed purely to collect statistics and provide resource sharing along with enhancing the flexibility of the system as a tool. In this way resources shared between machines and synchronization primitives needed to coordinate processes on machines across the network, can be found in one centralized location.

D. System Modifications

Every time a scenario updates its local clock or calculates a new event time it enters the SIMMON monitor and

executes the algorithm that was explained earlier. Since this algorithm is shared by all processes only one copy of the code is needed and a substantial savings in code is gained in local process or scenario size. Modifications to the system's scenarios are easy to manage in this manner. The local data needed for the event calculations and distributed synchronizations are also maintained within the monitor removing all of these responsibilities from the logic of the simulating processes.

Since the SIMMON monitor was originally an event-driven system the code that manipulates the scenarios will need to be replaced by the algorithm's method of synchronization. Processes will be delayed in the monitor until the communication sequence is completed, which allows it to update its local clock to the next event. To achieve the maximum amount of input/output from the system a communication process will be created within the monitor as a static process for every input and output line that crosses a machine boundary. This will allow each logical process a communication link process to, in effect, balance the work load.

The SIMMON monitor uses the master process concept for simulation initiation and termination. This concept needs to be extended to the design of the distributed network. During the initialization of the network, one master network process must be created with slave initialization process on

the corresponding SIMMON hosts. Upon initialization, these processes communicate shared resources, communicate process and other data needed for the global system such as termination time. This will add one more process to the others resident upon the respective nodes. The master process content will depend upon the model of the network itself. And will provide calls to the monitor to inform it of the master node, and the communication to support network initialization.

This design will deal with only the algorithm and the communication process alterations to SIMMON. Other changes in event scheduling will be dealt with in the actual implementation of the monitor, because not all system changes can be foreseen in this implementation design.

Upon termination the master process must initiate a file transfer of the statistics collected upon the other host nodes. This file transfer would just be another prefix call to the NADEX Operating System. This information is collected within one file resident on the master processes Host computer, providing a centralized location for statistic collection.

Communication Process

Upon initialization a communication process will be brought up to correspond to each input and output line on a

node. The algorithm that would achieve this communication interface would appear:

Communication process

Cycle;

FOR ALL I IN COMMUNICATION SET

DO;

IF I IN INPUT SET THEN

READ_PARM (I , tuple)

SIM.MESSAGE_BUFFER.PUT

IF I IN OUTPUT SET THEN

SIM.MESSAGE_BUFFER.GET

WRITE_PARM (I, tuple)

Forever

END

The scenarios of the envisioned system would contain just the logic to simulate the activities of the physical system being simulated. This can be done with "WAIT-EVENTS" and "SIGNAL-EVENT" accompanied with "WAIT-TIME" for the time update mechanism.

Notes on Performance

Overhead in the system is contributed to the communication system and the interaction of processes, and the synchronization of the processes. A problem with some distributed systems is that there is such a large volume of communication required, that the time required to run the distributed program is not significantly less than the time required to run a sequential program.

Empirical evidence [12] suggests that the null message approach to deadlock avoidance is expensive because of the large number of messages transmitted are null messages.

IV. Concluding Remarks

A survey of the area of distributed simulations has been the main thrust of this report. The computer industry is gearing up for the possible development and advancement within the distributed arena. A complete understanding of distributed systems is needed before the users of such systems can utilize these advantages to the utmost. This report surveyed the published material on distributed simulation with emphasis upon the link time algorithm. A great deal of emphasis is being placed upon distributed processing as local networks and multiprocessor systems are becoming more and more prevalent in industry. Using the algorithms presented within this paper, distributed systems can be made to perform tasks that in the past have been thought of as unmanageable, due to synchronization problems and communication protocol. Distributed tools such as the ones proposed within this paper are needed to bridge the "computer-user" software gap. Computer models are growing more complex as users are asking more from the computer simulation hosts. Their demands usually focus upon greater flexibility and faster turnaround. The proposed design would allow the user more freedom in the simulation of inherently parallel models.

It is the modeler that has the biggest factor in the modeling process. If the configured model does not uphold the rules of concurrent structuring, then little could be

expected from the increased speed and flexibility this design. This report contains two "link time" algorithms for possible implementation with existing tools at Kansas State University. These algorithms should be viewed as possible alternatives to the current simulation systems. Since concurrent Pascal lends itself nicely to "event-driven" systems and both of the existing tools (SIMMON and NADEX) now utilize C-Pascal as a host language, the modification to the systems should be just another step in the evolution of tools such as SIMMON and NADEX, and the development of tools for the users of distributed networks.

1. Young, R. and Wallentine, V. "The NADEX Core Operating System Services." Technical Report TR-791-11. Department of Computer Science, Kansas State University, Manhattan, Ks., November 1979.
2. Rochat, K. "User Control of Software Configuration, Under The NADEX Operating System." Master's Thesis. Department of Computer Science, Kansas State University, Manhattan, Kansas.
3. Brinch Hansen, Per. The Architecture of Concurrent Programs. Prentice-Hall, Inc., 1977. Englewood Cliffs, New Jersey, 07632.
4. Knuth, D. and McNeley, J.L. "SOL-A Symbolic Language for General Purpose System Simulation." IEEE Trans. on Computers (Aug., 1964).
5. Gordon, G. The Applications of GPSS V to Discrete System Simulation. Englewood Cliffs, New Jersey, Printice-Hall, 1975.
6. Kiviat, P. J.; Villaneuoa, R.; and Markowitz, H. M. The Simscript II Programing Language. Englewood Cliffs, NJ, Printice-Hall, 1968.
7. Vausher, J. G. and Dwal, P. "A comparison of Simulation Event List Algorithms." CACM Vol. 18, No. 4 (April, 1975).
8. Hoare, C. A. R. "Communicating Sequential Processes." Communications of the ACM 21. No. 8. (August 1978).
9. Chandy, K. and Misra, J. "Distributed Simulations: A Case Study in Design and Verification of Distributed Programs." IEEE Trans. on Software Engg., Vol. SE-5, No. 5. September, 1979.
10. Peacock, J. D. and Wong, J. W. and Manning, E. "Distributed Simulation Using a Network of Processors." Computer Networks, 3. (1979).

11. Balzer, R. "Ports-A method for Dynamic Interprogram Communication and Job Control." Proc. AFIPS Spring Joint Computer Conference 38. (1979), 489-489.
12. Chandy, K. and Misra, J. "Asynchronous Distributed Simulation via a Sequence of Parallel Computations". Communications of the ACM, Vol. 24, No. 4, (April 1981).

DESIGN OF A DISTRIBUTED SIMULATION TOOL DESIGN

by

HARRY L. PHELPS

B.S. CS, Kansas State University, 1979
B.S. CPT, Kansas State University, 1981

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

Year: 1981

ABSTRACT

This report describes a network simulation system design using a link time method of synchronization. The design utilizes two existing tools that were developed at Kansas State University: SIMMON, a Simulation facility and NADEX, a network operating system. The topic is presented in this fashion:

- 1) An overview is given of simulation in a distributed environment.
- 2) An in-depth description of the algorithm used in the system design is presented.
- 3) A description of a possible implementation of the algorithm is given using SIMMON and NADEX as host environments.

The report contains a tutorial on distributed simulation along with descriptions of modifications to SIMMON that allows it to perform in this distributed environment. The goal of the work described in this report is to present an alternative to sequential simulations which don't provide the capability for concurrent processing multiple processor facilities. This proposed system would allow the user to decompose a modeled system among a network of computers so that the workload could be distributed among them.