Distributed Simulation: A Case Study in Design and Verification of Distributed Programs

K. MANI CHANDY AND JAYADEV MISRA, MEMBER, IEEE

Abstract—The problem of system simulation is typically solved in a sequential manner due to the wide and intensive sharing of variables by all parts of the system. We propose a distributed solution where processes communicate only through messages with their neighbors; there are no shared variables and there is no central process for message routing or process scheduling. Deadlock is avoided in this system despite the absence of global control. Each process in the solution requires only a limited amount of memory. The correctness of a distributed system is proven by proving the correctness of each of its component processes and then using inductive arguments. The proposed solution has been empirically found to be efficient in preliminary studies. The paper presents formal, detailed proofs of correctness.

Index Terms—Concurrent processes, distributed systems, performance, program proving, simulation.

I. Introduction

A. Distributed Programs

PROGRAMS which consist of two or more cooperating processes which communicate with each other *exclusively* through messages will be called *distributed* programs. In particular, processes in a distributed program cannot share variables; every variable in a distributed program must be local to a single process. Furthermore, *control in distributed programs* is *distributed*; there are no central processes through which messages are routed, nor are there control processes which direct the operation of other processes. If there are enough processors available, a distributed program may be run on a loosely coupled system by running one process on each processor. Distributed programs are likely to be increasingly common in the future due to the rapid decline in the cost of processing. In this paper, we develop a distributed algorithm for solving a class of simulation problems.

B. An Overview

The Physical System or the System to be Simulated

We consider physical systems in which processes communicate exclusively through messages. A process may decide to send a message at any arbitrary time $t > 0$. Whether a message is sent out at t or not depends only on the messages received by the process up to t. The contents of the message sent out by a process at t, if any, also depend only on the messages received by that process up to t. Note that the output messages of a process may depend both on the content and on the times at which messages were received by the process. Examples of such systems are job shops (where messages are jobs), data base systems, and communications networks.

The Logical System (Simulator)

Each process in the physical system is simulated by a separate logical process. We use the term LP for logical process and PP for physical process. The logic of an LP depends only upon the PP that it is simulating; it is independent of the rest of the physical system. We place no restriction on the logic of any PP provided it can be simulated. There is a communication line from the i*th* LP to the j*th* LP in the logical system (simulator) if and only if the i*th* PP sends messages to the j*th* PP, in the physical system. Hence there is no central process which controls synchronization among the various processes. Note that this implies there is no variable, such as simulated time, shared by all LP's.

Asynchronous Behavior of the Logical System

The logical system is asynchronous: we place no restriction on processor speeds. The key to achieving asynchronism in the logical system is the encoding of physical time as a part of messages communicated among LP's. A message m sent by the i*th* process to the j*th* process at time t in the physical system is simulated by a message (t, m) from the i*th* LP to the j*th* LP in the logical system.

The communication in the logical system can be based on any protocol; in particular we consider a very simple protocol which is easily implemented. This protocol was suggested in a pioneering paper by Hoare [9] on distributed programming.

Simulation Through Encapsulation

The logic of each LP, in our solution, consists of two parts: simulation of the corresponding physical process (determining what messages go from one physical process to another, at certain times) and communication with other LP's (sending tuples of the form (t, m) along certain selected lines and selecting lines on which messages should be received next). The simulation of a physical process by its corresponding logical process is totally independent of the rest of the system. In

Manuscript received November 27, 1978; revised May 22, 1979. This work was partially supported by the National Science Foundation under Grant MCS 77-09812.

The authors are with the Department of Computer Sciences, University of Texas, Austin, TX 78712.

0098-5589/79/0900-0440$00.75 © 1979 IEEE

fact, if the physical process were a computing process, we could create a logical process by calling upon the physical process (with very slight changes in interface) to produce messages. We call this notion *encapsulation*; this allows us to study the distributed aspect of simulation (message communication, etc.) independent of the characteristics of the physical processes being simulated. In particular, we can use existing simulators of physical processes in developing distributed simulations (see Appendix).

Deadlock Prevention without Global Control

A major problem in concurrent programming is to avoid deadlock. In our system, deadlock is avoided without global control and in a manner independent of the structure of the network. Each process in the logical system can be thought of as an elementary building block and arbitrarily complex structures can be created by connecting elementary building blocks together. Absence of deadlock in the resulting arbitrary structure results from the logic of each process even though each process's logic is independent of the overall structure of the network.

Minimal Memory Requirements

Any simulation algorithm must need at least as much memory as is required by the corresponding physical process; in particular if the physical process produces its outputs based on all inputs received so far, the simulator cannot in general avoid storing the entire history of inputs. In addition to the memory required for simulation of the physical process, every simulator requires some extra storage, such as for the "event-list" in conventional simulation. In the case of distributed simulation, extra storage is often required, particularly for avoiding deadlock. It is easier [15], [16] to obtain deadlock free distributed programs if the LP's have infinite amounts of memory for storing arbitrary numbers of messages. Our solution requires a *bounded amount of extra storage* for each LP.

Correctness without Global Control

The correctness of the overall system is deduced even though the logic of each LP depends only on the physical process that it simulates and is otherwise independent of the system.

Proving Distributed Program Properties

All our proofs regarding the behavior of the logical system have the following structure:

first: prove a property of each individual process in the system assuming that all other processes in the system satisfy that property;

second: use inductive arguments to prove that the system as a whole satisfies this property.

Evaluation of Performance

The performance of our distributed program is discussed in [13]. A problem with some distributed systems is that there is such a large volume of communication traffic between processes relative to the amount of computation required, that the time required to run the distributed program is not significantly less than the time required to run a sequential program. We show that the time required to run our distributed program for the specific problem of queuing network simulation is generally less than the time required to run corresponding sequential programs. This efficiency is achieved since there is no global process which could be a bottleneck.

C. Related Work[1]

The work reported here has been influenced by many other pieces of work. Our approach borrows concepts from data flow architectures proposed by Dennis [3], [4], problem decomposition schemes of Dijkstra [5] and Wirth [14], and interprocess communication of Hoare [9]. A subsequent paper uses Hoare's elegant notation to describe solutions for a specific simulation problem.

An attractive proof technique for general parallel programs has been presented by Owicki and Gries [11]. Since we work with much more restricted programs our proofs are much simpler; in particular our concurrent processes cannot interfere with each other. Our proof of correctness has similarities with Patil's [12] work on determinism; however our absence of deadlock proof is novel.

Hoare and Kaubisch [10] have proposed a method of distributed simulation using a central clock. Peacock, Wong, and Manning [16] and Holmes [15] propose schemes for distributed simulation without using central clocks on specific existing architectures. Both the schemes require buffers of unbounded length to guarantee absence of deadlock.

A more detailed tutorial version of this paper including several examples may be found in [17].

D. Overview of the Paper

In Section II, the class of physical systems is described, for which distributed simulation algorithms will be given. In Section III, the algorithm for each logical process in the distributed simulation solution is given. An explanation of the algorithm and some of its properties are described intuitively in Section IV. Section V contains some preliminary empirical results on the performance of the algorithm. The reader can get an intuitive idea of the algorithm by studying these four sections. Section VI contains proofs about correctness, absence of deadlock, and termination properties of the distributed solution. The reader, who is not interested in proofs and proof methodology, may skip Section VI. The Appendix

[1] Recently, it has been brought to our attention that R. E. Bryant has independently developed the idea of distributed simulation. His work appears in "Simulation of packet communication architecture computer systems," MIT/LCS/TR-188, MIT, Nov. 1977. The most important difference between our work and his work is that he assumes that a logical process is able to output a message whenever it wants to. This implies that 1) processes would require buffers of unbounded length and 2) the only possibility of deadlock arises when all processes in a loop wait for input, which can be easily shown to be impossible. We however require that a logical process must wait to input/output to another process until the second process is ready to output/input. This leads to memory requirements which are bounded. Furthermore our algorithm is constructed differently and the absence of deadlock proof is considerably more involved.

contains a detailed description of the encapsulation of physical processes.

II. The Physical System

A. Introduction

We shall simulate a system of N processes which communicate with each other exclusively through messages. Let the processes be indexed $1, \cdots, N$. This system will be represented by a directed graph G of N vertices $\{v_1, \cdots, v_N\}$, where v_i represents process i. There will be an arc from v_i to v_j if and only if process i sends messages to process j. We will not allow a process to send messages directly to itself because that effect can be achieved by a process looking at its own computation.

At any given time t, a process i may decide to send a message to another process j; the decision to send the message and the message content are determined uniquely by process i's (internal) logic and by the messages it has received so far.

The actual time spent in transmitting a message from process i to process j is assumed to be zero. (If it is important to simulate the fact that process i takes some time $T > 0$ to send a message to process j, we shall assume that process i spends zero time in *initiating* the message send, and then spends T time units without carrying out any other function. Similarly if it is important to simulate the fact that process j takes some time $T > 0$ to receive a message, we shall assume that it spends zero time in *initiating* the receive and then spends T time units without carrying out any other function.)

A system is simulated for some time period $[0, Z]$ where Z is referred to as the *termination time.* During this time, any process sends a finite number of messages (or more accurately, initiates a finite number of message sends) to other processes. Let some process p_i send some other process p_j, messages at some instants $t_1, \cdots, t_K$, where

$$0 < t_1 < \cdots < t_K \leqslant Z \tag{1}$$

We restrict attention to systems in which some time must elapse after a process sends one message before it sends another. We do not lose generality due to this restriction because there do not exist systems in which a process i can generate and transmit two or more messages to another process j in arbitrarily small time. We assume that there exists a prespecified positive constant ϵ, such that

$$t_k - t_{k-1} > \epsilon \quad \text{for } k = 2, \cdots, K \tag{2}$$

Let m_k be the k*th* message sent from process i to process j, $k = 1, \cdots, K$. The stream of messages from process i to process j is described by the *tuple sequence for arc* (i, j):

$$s_{ij} = ((t_1, m_1), \cdots, (t_K, m_K)) \tag{3}$$

The *message history of arc* (i, j) at time $t \geqslant 0$, $h_{ij}(t)$ is defined to be the following initial subsequence of s_{ij}:

$$h_{ij}(t) = \begin{cases} (\ ) & \text{if } t_1 > t \\ s_{ij} & \text{if } t \geqslant t_K \\ ((t_1, m_1), \cdots, (t_k, m_k)) & \text{if } t_k \leqslant t < t_{k+1} \\ & \text{for } k = 1, \cdots, K-1 \end{cases} \tag{4}$$

Thus the message history of arc (i, j) at t is the sequence of all tuples (t, m) corresponding to messages sent at or before t; this history is a complete specification of messages transmitted along arc (i, j) at or before t. For convenience, we define $h_{ij}(t)$ to be ( ) for all t, if there is no arc (i, j).

We restrict attention to systems where the output of a process at time t depends solely upon the messages received by the process at or before t. All physically realizable systems meet this assumption.

From this restriction it follows that there exists a function f_{ij} such that

$$h_{ij}(t) = f_{ij}(t, h_{1i}(t), \cdots, h_{Ni}(t)) \tag{5}$$

B. Predicting Future Output from Current Histories

In some cases it is possible to deduce the histories of messages that will be sent by a process i to a process j up to some time t' solely from the histories of messages received by process i up to some earlier time t. In this case, messages (if any) sent by process i to process j in the interval (t, t') are independent of messages received by process i after t. We define *lookahead* for the arc (i, j) as $t' - t$. The value of lookahead depends only upon t, and the message histories obtained by process i at t. Formally, lookahead for arc (i, j) is the function:

$$L_{ij}(t, h_{1i}(t), \cdots, h_{Ni}(t)) = t' - t \tag{6}$$

We assume that the functions $L_{ij}(\ )$ are computable, and in the simulation we will compute this function for several values of t and h_{ki}. For brevity we shall not show the explicit dependence of L_{ij} on the input histories and we use the short form $L_{ij}(t)$.

t' is the point to which the output of process i to process j can be predicted given the input histories up to time t. Let t'' be the point to which this output can be predicted given the input histories up to some later time $t + \Delta$ (where $\Delta > 0$). We make the reasonable assumption that

$$t'' \geqslant t'$$

i.e.,

$$(t + \Delta) + L_{ij}(t + \Delta) \geqslant t + L_{ij}(t) \tag{7}$$

In other words, additional input information cannot reduce the point to which the output can be predicted.

Equations (5) and (6) imply that there exists a function F_{ij} where

$$h_{ij}(t + L_{ij}(t)) = F_{ij}(t, h_{1i}(t), \cdots, h_{Ni}(t)) \tag{8}$$

We assume that the F_{ij} are computable.

C. The Predictability Property

If the graph representing the system of processes has loops, the information input to a process i may be a function of the information output from that process. If the input to process i at t is a function of the output from it at t, and if the output from it at t is a function of the input to it at t, we have a circular definition where the information input to process i at t is a function of itself. To avoid such a situation we restrict attention to systems where in every loop there exists an arc (i, j) with

$$L_{ij}(t) > \epsilon \qquad (9)$$

for all t, where ϵ is the prespecified positive constant defined in (2). We call such arcs (i, j), *predictable arcs.*

Note that $L_{ij}(t)$ is a property of process i and the messages it receives, and is otherwise independent of the system. Lookahead is a *local* property. On the other hand predictability is a *system* property because it depends on how processes are configured into a system.

D. Source and Sink Processes

It is helpful to identify two special types of processes: *sources* and *sinks.* A source process is one which sends messages to other processes but does not receive messages from other processes. A sink process is one which receives messages but does not send any. We shall not consider processes which neither send nor receive messages.

III. The Logical System

A. Introduction

We are given a physical system of N processes. Graph G (see Section II) represents the topology of information flow between processes. We simulate this system by a network of N processes: $v_1, \cdots, v_N$. To distinguish between processes in the physical system and processes in the simulator we refer to the former as *physical* processes (PP) and to the latter as *logical* processes (LP); the *ith* LP simulates the *ith* PP. Information is transmitted from LP i to LP j if and only if PP i sends messages to PP j. For purposes of exposition it is convenient to assume the existence of an imaginary communication channel from LP i to LP j in such a case: this channel will be referred to as line (i, j).

LP's communicate exclusively be sending messages to one another; an LP does not have shared variables with any other LP. The communication in the logical system can be based on any protocol. In this paper we assume a very simple protocol: *a message is sent from LP i to LP j if and only if LP i is ready to send the message and LP j is ready to receive it.* We assume the existence of an implementation which ensures that messages are transmitted correctly between LP's. These protocols were presented by Hoare [9] who developed an elegant language to describe communication between processes.

The only aspect of the physical system that is of interest to us is the sequence of messages communicated between physical processes. *Each event 'PP i sends PP j a message m at time t' is simulated by LP i sending LP j a 2-tuple: <t, m>.* All messages between LP's will be 2-tuples, <t, m> where $0 \leqslant t \leqslant Z$ and m is either a message sent at time t in the physical system or is a special symbol: NULL, which does not occur in the physical system.

Definition: For a tuple $<t_k, m_k>$, t_k is called the *t-value* of the tuple and m_k is called the *m-value* of the tuple.

B. Description of Tuples Transmitted by Logical Processes

Let s_{ij} be the tuple sequence describing messages sent from PP i to PPj. [See (3).] Let the sequence of messages sent from LP i to LP j, in the simulation, be

$$S_{ij} = ((T_1, M_1), \cdots, (T_l, M_l)) \qquad (10)$$

We will establish in Section VII that (11)-(15) hold; these equations are concerned with the correctness of the simulator.

There exist constants A and B (independent of Z) such that

$$l \leqslant AZ + B \qquad (11)$$

That is, the number of tuples sent on any line in the simulator is bounded by a linear function of Z.

$$0 \leqslant T_1 < T_2 < \cdots < T_l \leqslant Z \qquad (12)$$

That is, the sequence of tuples sent from one LP to another are in strictly increasing order of t-values.

$$(t_k, m_k) \in s_{ij} \Rightarrow (t_k, m_k) \in S_{ij} \qquad (13)$$

That is, every tuple in s_{ij} is also in S_{ij}; thus we will simulate every message from PP i to PP j.

$$(T_k, M_k) \in S_{ij} \Rightarrow [(T_k, M_k) \in s_{ij} \text{ or } M_k = \text{NULL}] \qquad (14)$$

That is, the only tuples in S_{ij} which are not in s_{ij} have NULL m-values. LP i sends LP j a tuple <t, m> where m is not NULL if and only if PP i sends PP j the message m at time t; in addition to these tuples LP i may send LP j an arbitrary (though bounded) number of tuples with NULL m-values.

$$T_l = Z \qquad (15)$$

That is, the last tuple sent from one LP to another must have a t-value equal to the termination time.

C. Specification of Tuple Histories

We shall define a history of tuples along line (i, j) in the logical system in a manner analogous to the definition of the message history for arc (i, j) in the physical system [see (4)].

Let S_{ij} be a sequence of tuples as defined in (10)

$$H_{ij}(t) = \begin{cases} ( ) & \text{if } t < T_1 \\ ((T_1, M_1), \cdots, (T_r, M_r)), & \text{where} \\ T_r \leqslant t & \text{and either } r = l \text{ or } t < T_{r+1}. \end{cases} \qquad (16)$$

$h_{ij}(t)$, the history of messages from PP i to PP j at time t [see (4)] is obtained by removing all tuples with NULL m-values from $H_{ij}(t)$. Hence, when LP j receives a tuple $<T_k, M_k>$ from LP i, LP j knows that it has obtained a complete specification of the message history from PP i to PP j at T_k; note that this statement is true regardless of whether M_k is NULL. Thus the purpose of tuples with NULL m-values is to extend the times up to which message histories are known.

D. Predicting Output Information from Specifications of Message Histories

Define T_{ki} as the t-value of the last tuple transmitted along line (k, i) and if no tuple has been transmitted along that line set T_{ki} to 0. We shall refer to T_{ki} as the clock value of line (k, i). The clock value of line (k, i) is the point in physical time up to which the line (k, i) has been simulated by LP k and LP i.

$$\text{Let } TIN_i = \underset{k}{\text{minimum}}\, T_{ki} \qquad (17)$$

Then LP i knows that it has a specification of all inputs to PP i up to time TIN_i and hence LP i can determine [see (8)] all the messages sent from PP i to PP j up to time

$$\mathrm{TOUT}_{ij} = \mathrm{TIN}_i + L_{ij}(\mathrm{TIN}_i) \tag{18}$$

From (7) it follows that increasing TIN_i will not decrease TOUT_{ij}. Note that TIN_i and TOUT_{ij} are undefined for source and sink, respectively.

E. Input/Output (I/O) Operations in the Logical System

We are only interested in messages transmitted in the physical system and in tuples transmitted in the logical system. Hence we focus attention on the I/O's carried out by each LP. The simulation can be described by sequences of I/O's.

An I/O operation is the receiving/sending of *one* tuple along a line. Recall that an I/O operation takes place only when an LP at one end of a line wants to receive/send a tuple along that line and the LP at the other end of the line is ready to send/receive a tuple along that line. Note that an LP may have to wait to receive/send a tuple if the LP at the other end of the line is not ready to send/receive. In some cases an LP may wish to carry out (and if necessary, wait) simultaneously for I/O operations on several lines. In such cases, the individual I/O operations are carried out in a nondeterministic order. The composite operation consisting of I/O operations on one or more lines, in the above manner, is called a *parallel I/O operation.* The parallel I/O operation is completed when all its component individual I/O operations are completed.

Simulation Overview

After the initialization, the simulation algorithm for each LP i repeats the following sequence of steps until *termination.*

1) (Selection): Select the set of lines NEXT_i, on which the I/O operations will next be carried out in parallel. NEXT_i may contain only input lines or only output lines or both.

2) (Computation): For every output line in NEXT_i, determine the next tuple to be transmitted along that line.

3) (I/O operation): Carry out the parallel I/O operation for all lines in NEXT_i.

For the source LP, there are no input lines and hence the sequence of output messages is determined *a priori.* The source follows steps ①, ②, ③ exactly like any other LP.

1) Selection

Recall from the definition of TOUT_{ij} [(18)] that it is possible to predict the output tuples on the line (i, j) with t-values up to TOUT_{ij}. If,

$$\mathrm{TOUT}_{ij} > T_{ij}$$

then LP i can send more tuples along line (i, j) without waiting for additional input; such an output line is said to be *open.* Let

$$T_i = \underset{\text{for all } j,k}{\mathrm{minimum}} (T_{ij}, T_{ki}) \tag{19}$$

T_i will be called the *clock value* of LP_i; this value is the minimum t-value over all incident lines on LP i; T_i represents the point in time up to which LP i has simulated PP i. Let

$$\mathrm{NEXT}_i = \{k \mid T_{ki} = T_i\} \cup \{j \mid T_{ij} = T_i \text{ and } \mathrm{TOUT}_{ij} > T_{ij}\} \tag{20}$$

Thus NEXT_i is the set of all input and open output lines with clock values equal to the clock value of LP i. LP i will carry out I/O operations on all the lines in NEXT_i.

The t-value of a line is the point in (physical) time to which the line has been simulated by LP's at both ends. The LP clock value is the point in (physical) time to which the corresponding PP has been simulated. The goal of the algorithm is to move the LP clock value forward. This goal forces us to wait on at least all the lines in NEXT_i. We show later that this definition of NEXT_i avoids deadlock. Furthermore, we show in the Appendix that memory requirement is bounded for each LP, due to this definition of NEXT.

Note that for an output line (i, j) for which $\mathrm{TOUT}_{ij} = T_{ij}$ the line has already been simulated up to TOUT_{ij} and hence no further tuples can be sent along that line until TOUT_{ij} is increased. Hence such lines cannot be included in NEXT_i.

2) Computation

The following algorithm shows the steps taken in computing the next tuple to be sent along an output line (i, j), where $j \in \mathrm{NEXT}_i$. Recall that $j \in \mathrm{NEXT}_i$, where (i, j) is an output line, if $T_{ij} < \mathrm{TOUT}_{ij}$.

Algorithm (III-E-2):

{Given line (i, j), $j \in \mathrm{NEXT}_i$, compute the next tuple (t, m) to be output along (i, j)}.

1) From $\mathrm{TIN}_i (= \min_k T_{ki})$ and all the input histories up to TIN_i, compute $h_{ij}(\mathrm{TOUT}_{ij})$ by (8), i.e., $h_{ij}(\mathrm{TOUT}_{ij}) = F_{ij}(\mathrm{TIN}_i; h_{1i}(\mathrm{TIN}_i) \cdots, h_{Ni}(\mathrm{TIN}_i))$.

This is computable since F_{ij} is computable. Details of computing $F_{ij}(\ )$, which amount to simulating the corresponding physical process, are explained in the Appendix. The correct computation of F_{ij} implies that $h_{ij}(\mathrm{TOUT}_{ij})$ is correct if $h_{1i}(\mathrm{TIN}_i), \cdots, h_{Ni}(\mathrm{TIN}_i)$ are correct.

We now consider the following two cases for computing the next tuple (t, m).

Case 1: $h_{ij}(T_{ij}) \neq h_{ij}(\mathrm{TOUT}_{ij})$

This implies that PP i sends a message on line (i, j) in the interval $(T_{ij}, \mathrm{TOUT}_{ij}]$.

If PP i sends out the sequence of messages $m_1, m_2 \cdots$ at times $t_1, t_2 \cdots$ in the interval $(T_{ij}, \mathrm{TOUT}_{ij}]$, then set $(t, m) := (t_1, m_1)$.

Case 2: $h_{ij}(T_{ij}) = h_{ij}(\mathrm{TOUT}_{ij})$

This implies that PP i does not send any message on line (i, j) in the interval $(T_{ij}, \mathrm{TOUT}_{ij}]$. Set $(t, m) := (\mathrm{TOUT}_{ij}, \mathrm{NULL})$.

Note:

1) $T_{ij} < t \leqslant \mathrm{TOUT}_{ij}$

2) Following the output operation (discussed next), T_{ij} would be increased to t computed in this step. T_{ij} is increased to TOUT_{ij} following output of a tuple with NULL m-value.

3) Case 1) computes only (and all) non-NULL m-values and Case 2) computes only (and all) NULL m-values.

3) I/O Operation

This step of the algorithm consists of waiting in parallel to input/output along the selected lines in NEXT_i and updating the clock value and value of the last message on every such line. If (t, m) is the tuple received or sent along line (i, j) as a result of the I/O operation, then the corresponding line clock value is increased to t and the value of the last message is updated to m, immediately following transmission of this tuple.

4) Initialization

The local data maintained by LP i are the following: (T_{ki}, M_{ki}), for each input line (k, i) denotes the tuple last received on this line at the beginning of the selection step. Initially this is set to (0, NULL), which indicates that the line has been simulated up to time 0. Similarly (T_{ij}, M_{ij}), maintained for each output line (i, j), denotes the tuple last transmitted on that line at the beginning of the selection step; this is initially (0, NULL). Let ($NEWT_{ij}$, $NEWM_{ij}$) denote the next tuple to be transmitted along line (i, j); these temporary variables are maintained for every output line. Algorithm III-E-2 computes ($NEWT_{ij}$, $NEWM_{ij}$) whenever $j \in NEXT_i$. Furthermore, T_i, the clock value of LP i, is also maintained; this is set to 0 initially.

5) Termination

Since it is required that the simulation should proceed up to Z, the LP should repeat the steps of Section III-E (selection, computation, and I/O operation) as long as the clock value of the process is less than Z. For simplicity, it is preferable to have the clock value of every line to be Z at the termination of the simulation. There is a possibility that the next tuple (t, m) to be output on some line (i, j) may have $t > Z$. In such a case, the LP outputs (Z, NULL) along that line. This is a correct output, since in the physical system no message is sent along (i, j) in the interval (T_{ij}, Z). Furthermore, this must be the last output along that line, since the line will never be selected again.

F. Summary of the Algorithm for Logical Process i

```
Algorithm (III-F):

{Initialization}
 $T_{ij}$ := 0; $M_{ij}$ := NULL, for every j;
 $T_{ki}$ := 0; $M_{ki}$ := NULL, for every k;
 $T_i$ := 0;
 while $T_i < Z$ do
     {selection}
       $NEXT_i := \{k \mid T_{ki} = T_i\} \cup \{j \mid T_{ij} = T_i < TOUT_{ij}\}$;
     {computation}
       for every output line (i, j) where $j \in NEXT_i$ do
           compute next output tuple ($NEWT_{ij}$, $NEWM_{ij}$); {algorithm (III-E-2)}
           if $NEWT_{ij} > Z$ then ($NEWT_{ij}$, $NEWM_{ij}$) := (Z, NULL) endif
       endfor;
     {I/O operation}
       Do the operations 1 and 2 in parallel:
      1. for every output line (i, j), $j \in NEXT_i$, wait to output
         ($NEWT_{ij}$, $NEWM_{ij}$) and set ($T_{ij}$, $M_{ij}$) to ($NEWT_{ij}$, $NEWM_{ij}$)
         immediately following transmission of the tuple and
      2. for every input line (k, i), $k \in NEXT_i$, wait for input and then
         set ($T_{ki}$, $M_{ki}$) equal to the incoming tuple on line (k, i);
     {compute $T_i$: the clock value of the process}
       $T_i := \underset{j,\,k}{\text{minimum}}\ (T_{ki}, T_{ij})$
 endwhile
```

IV. Intuitive Explanation of the Algorithm

We will show in a later section, that any network of logical processes each of which executes the algorithm of Section III-F, is correct, deadlock free, and terminates properly. Intuitive definitions of these terms and some of the difficulties in proving these facts for arbitrary networks are sketched out in this section.

We say that a sequence of tuples is *chronological* if successive t-values strictly increase. If the logical system correctly simulates the physical system, it must produce a chronological sequence of tuples whose m-values are non-NULL. It is not obvious that the proposed algorithm does so; we shall formally prove this property in Section VI. The sequence of tuples produced on line (i, j), S_{ij}, is *correct* if it matches the corresponding sequence s_{ij} in the physical system, upon removing the tuples with NULL m-values from S_{ij}. The proof of correctness of the logical system will be established in two steps: we will first show that every individual LP is correct, i.e, it produces correct output on every line given correct input on every line at any point in simulation; we next show that any arbitrary interconnection of correct LP's results in a correct overall system.

We also show that our system is *deadlock-free*, i.e., irrespective of the sequence in which various I/O operations in the system are executed, there is always an I/O operation that can take place; in other words there will never be a set of processes which are waiting for each other. This problem, a serious one in any distributed program, was avoided by careful selection of the lines on which the next I/O operation may take place.

Absence of Deadlock

An intuitive explanation of the proof of absence of deadlock is as follows. Suppose deadlock occurs, then consider

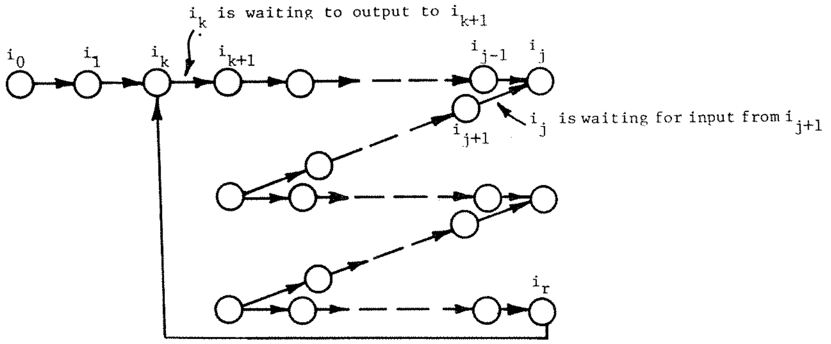

i_k is waiting to output to i_{k+1}
i_0
i_1
i_k
i_{k+1}
i_{j-1}
i_j
i_{j+1}
i_j is waiting for input from i_{j+1}
i_r

Fig. 1. A pseudoloop of waiting processes in case of a deadlock.

any LP i_0 that is waiting to carry out either an input or output operation; let i_1 be the LP for which it is waiting. Similarly let i_2 be the LP for which i_1 is waiting. In general let i_{q+1} be the LP for which i_q is waiting. Continuing in this manner, we must repeat an LP since the network is finite. It is instructive to consider the sequence of LP's so computed and their waiting relationships. There is a line (i_q, i_{q+1}) if i_q is waiting to output to i_{q+1}; there is a line (i_{q+1}, i_q) if i_q is waiting to input from i_{q+1}. The set of LP's and the lines as given above constitute a *pseudoloop* of the form in Fig. 1.

We will show that $T_{i_q} \geqslant T_{i_{q+1}}$. Hence all the process clock values in this pseudoloop must be identical. Furthermore, we will show that if i_q is waiting to *output* to i_{q+1} then $T_{i_q} > T_{i_{q+1}}$. Hence no such line (i_q, i_{q+1}) can exist. Thus the pseudoloop can only be a loop in which every LP i_q is waiting to input from i_{q+1}. The loop must contain a predictable line (Section II-D) and we will show that in such a case, waiting of the above form is impossible because $T_{i_q} > T_{i_{q+1}}$ in this case.

Termination

It may seem that a proof of correctness of output and a proof of absence of deadlock are sufficient to establish the correctness of the entire logical system. This is not so: it is likely that the simulation may never terminate if tuples are output with an arbitrarily small increase in t-values. In particular, in the example of Fig. 2, an infinite sequence of tuples can be output by process 1, each of which has a t-value strictly less than 1. This possibility arises since we allow t-values to be arbitrary real numbers, and an unbounded number of NULL m-values.

Assume that LP 1 outputs $(t_i + (1/2^i)$, NULL), for the *i*th output tuple and LP 2 simply transmits (t, m) from input to output. Then the successive t-values on line (1, 2) are 1/2, $1/2 + 1/4 \cdots$ and hence can never reach any $Z \geqslant 1$.

We show that such a situation is impossible and that every simulation will terminate in a finite number of steps.

Reason for NULL m-values

It is not obvious from the algorithm why a NULL m-value is useful: whether it is for efficiency alone (to move the clock value of the line as far as possible) or strictly required for correct operation of the system. We show in the example of Fig. 3 that deadlock is possible in the logical system if no tuples with NULL m-values are output. In fact deadlock is possible even in an acyclic network in such a case.

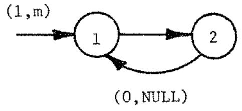


Fig. 2. A simulation that never terminates.

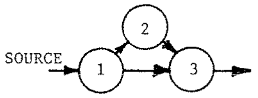


Fig. 3. An acyclic network that may deadlock.

Assume that in the physical system, the source produces messages m_1, m_2, m_3 at times 50, 100, 150; PP 1 outputs messages m_1, m_2, m_3 to PP 2 at times 50, 100, 150. PP 2 processes these messages and outputs m_1, m_2, m_3 to PP 3 at times 55, 105, 155. We show that if no tuples with NULL m-values are sent to LP 3 from LP 1 then the logical system deadlocks. Consider the sequence of steps in the logical system.

1) Source outputs $(50, m_1)$ to LP 1.
2) LP 1 outputs $(50, m_1)$ to LP 2.
3) LP 2 outputs $(55, m_1)$ to LP 3.
{LP 3 is now waiting for input from LP 1 alone}
4) Source outputs $(100, m_2)$ to LP 1.
5) LP 1 outputs $(100, m_2)$ to LP 2.
6) LP 2 *waits* to output $(105, m_2)$ to LP 3.
7) Source outputs $(150, m_3)$ to LP 1.
8) LP 1 *waits* to output $(150, m_3)$ to LP 2.

At this point LP 3 is waiting for LP 1 which is waiting for LP 2 which is waiting for LP 3.

This deadlock is avoided in our algorithm by LP 1 sending (50, NULL), (100, NULL), (150, NULL), etc., to LP 3.

What Happens When the Physical System Deadlocks?

It is interesting to note that the logical system *never* deadlocks: when the physical system deadlocks, the logical system continues computation by transmitting tuples with NULL m-values only and increasing t-values. This correctly simulates the corresponding physical situation in that while the time is increasing, no messages are being transferred in the physical system. Ultimately, the simulator will terminate with the clock value of every line at Z.

Choice of NEXT$_i$

The selection of lines NEXT$_i$, on which the next set of parallel operations are carried out in LP i, is crucial to the memory requirement of LP i and the absence of deadlock in the system. If waiting is permitted on a different set of lines, which does not include NEXT$_i$ as a subset, it can be shown that deadlock may arise. Hence LP i must wait on *at least* all the lines in NEXT$_i$.

If waiting is permitted on a superset of NEXT$_i$, for instance, on all input and output lines in each I/O operation step, then memory requirements become severe.

Taking Snapshots in a Distributed Simulation

Another interesting aspect of the algorithm is that a "snapshot" taken in the logical system does not in general correspond to any snapshot in the physical system. This is because the different lines have different line clock values, in general, indicating that they have been simulated up to different time instants. The proposed algorithm differs radically in this respect from conventional simulation algorithms. Note that it is possible to obtain a snapshot of our distributed system in the following way: each LP takes a snapshot of itself when its t-value is equal to the snapshot time t; the composite of all these process snapshots is the system snapshot at t. Use of snapshots in computing queuing statistics is described in [1].

V. Performance Analysis

We are in the process of empirically evaluating the performance of the suggested algorithm. In the absence of a distributed system of CPU's, we have been forced to simulate the logical system on available uniprocessor systems at the University of Texas. The physical system consists of a queuing network whose elements are sources, sinks, forks, merges, and queues. The logical process corresponding to fork or merge was assumed to take a third as much time as a queue process. A more efficient variant [13] of the algorithm was coded in Pascal. Preliminary performance results appear below.

Case 1) A Tandem Sequence of N queues:

Empirical observation:

$$\frac{\text{Turnaround time with N processors}}{\text{Turnaround time with 1 processor}} \cong 1/\text{N}.$$

Empirical studies were conducted for N = 2, 3, and 4, with 1000 jobs, which confirm this formula.

Case 2) Physical System Shown in Fig. 4:

Empirical data were generated where each branch of the fork was chosen with equal probability and 1000 jobs were run through the system.

Empirical observation:

$$\frac{\text{Turnaround time with 6 processors}}{\text{Turnaround time with 1 processor}} = \frac{2055}{7000}.$$

These results are encouraging. Thus turnaround time is reduced to 30 percent of its value by multiprocessing. Ideally, turnaround time should be reduced to $\frac{1}{6}$ of its value since six processors are being used; in this sense, the utilization of processors is low. However, as mentioned earlier, the increasing availability of processors makes it inappropriate to consider processor utilization as the key performance metric. Turnaround time is the only metric of interest.

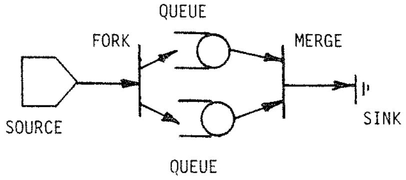


Fig. 4. An example of a physical system.

Fifteen other loop-free examples have been evaluated [13] which show that the distributed algorithm is significantly faster than the conventional sequential one. The examples are not included here for purposes of brevity. Evaluation of networks with loops is proceeding.

VI. Proof of Correctness of Simulation

In order to show that the proposed network of LP's correctly simulates the physical system, we have to prove (11)-(15). We divide the proof into the following four parts: 1) chronology of the tuple sequence, 2) correctness of every tuple sequence at any point in simulation, 3) absence of deadlock, and 4) termination of simulation. Each of these four proofs has two parts to it: proving certain facts about an individual LP from algorithms in Sections III-E-2 and III-F and then proving the desired property of the system based solely on the properties proved about individual LP's.

This proof methodology of dividing the proof into a *local proof* about individual LP's and a *global proof* based only on the properties shown in the local proofs has the following important implications.

1) The algorithm for one or more LP's could change without affecting the global proof and hence global correctness, provided the new algorithms have the same properties (as the old ones). Hence redesign of the system is straightforward.

2) The task of proving is considerably simplified in that internal details of processes are unnecessary for proving global properties. Local proofs abstract the relevant properties which are necessary for global proofs.

3) In the class of distributed programs that we have considered, correctness and chronology follow simply by showing that every individual process is correct and has the desired chronology properties. For a large class of properties, it is possible to show using induction that if individual processes possess these properties, then the system as a whole possesses these properties.

A. Chronology

We show that every sequence of tuples transmitted along any line has strictly increasing t-values, i.e., the sequence is *chronological.*

1) Process Chronology

Theorem 1: Consider any LP i at any point in the simulation. If every input sequence to i is chronological, at that point, then every output sequence is also chronological.

Proof: We show that for any output line i, j of i, (NEWT$_{ij}$,

$NEWM_{ij}$) computed in the computation step of the algorithm in III-F must have $NEWT_{ij} > T_{ij}$. t computed for line (i, j) in algorithm III-E-2 has, from the note, $t > T_{ij}$. Furthermore, $T_{ij} = T_i$ since $j \in NEXT_i$ and $T_i < Z$ from the loop iteration condition in algorithm III-F. The computation step sets $NEWT_{ij}$ to minimum (t, Z). Since both of these exceed T_{ij}, $NEWT_{ij} > T_{ij}$ following the computation step. Following the I/O operation step, T_{ij} equals $NEWT_{ij}$ and hence every output sequence is chronological.

Note: This proof shows that any output sequence is chronological even if the input sequence is not chronological. This may be used to prove system chronology (Theorem 2) trivially. We, however, prefer to use the weaker property of the processes stated in Theorem 1—this will permit us to interconnect any network of processes satisfying this weaker property and still achieve system chronology.

2) System Chronology

We assume that the source produces a chronological sequence of tuples.

Theorem 2: At any point in the simulation, the sequence of tuples along any line is chronological.

Proof: This theorem may be proven by induction on the number of times that an I/O operation takes place in the system. The proof is analogous to that of system correctness (Theorem 4) and hence is not repeated here.

B. Output Correctness

We show that the sequence of tuples transmitted along any line, at any point in the simulation is correct.

Recall definitions of $h_{ij}(t)$ [(4)] and $H_{ij}(t)$ [(16)]. Define $H_{ij}(t)$ to be correct with respect to $h_{ij}(t)$, denoted by $H_{ij}(t) \equiv h_{ij}(t)$, if and only if,

$$(t_k, m_k) \in h_{ij}(t) \Rightarrow (t_k, m_k) \in H_{ij}(t) \tag{21}$$

$$(T_k, M_k) \in H_{ij}(t) \Rightarrow (T_k, M_k) \in h_{ij}(t) \text{ or } M_k = \text{NULL} \tag{22}$$

Note that (13)-(15) merely assert that

$$H_{ij}(Z) \equiv h_{ij}(Z), \text{ for every line } (i, j).$$

1) Correctness of Process Output

The following theorem establishes the correctness of the output of an LP at any point in the simulation given that it has received correct inputs up to that point.

Theorem 3: (Correctness of Process Output)

Consider any LP i at any point in simulation. Assume that it has received correct input on every line up to that point; i.e., for every k,

$$H_{ki}(T_{ki}) \equiv h_{ki}(T_{ki}).$$

Then every output sequence produced so far is correct; i.e., for every j,

$$H_{ij}(T_{ij}) \equiv h_{ij}(T_{ij}).$$

Proof: We use induction on the number of iterations e of the loop in algorithm III-F.

When e = 0, $T_{ij} = 0$ for all j (initial conditions).
$H_{ij}(0) = (\ )$, from (16).
$h_{ij}(0) = (\ )$, from (1) and (4).
Hence the theorem holds.

Now assume that the theorem holds after e iterations. Consider the program point immediately following the computation step in algorithm III-F at the (e + 1)*th* iteration:

$H_{ki}(T_{ki}) \equiv h_{ki}(T_{ki})$ for all k, from conditions of the theorem.
$H_{ij}(T_{ij}) \equiv h_{ij}(T_{ij})$, for all j, from the induction hypothesis.

For any output line (i, j), we consider the following three cases.

Case 1: $h_{ij}(NEWT_{ij}) = h_{ij}(T_{ij})$.

Then $(NEWT_{ij}, NULL)$ is output along line (i, j) and T_{ij} is set to $NEWT_{ij}$ following the I/O operation. This follows directly from algorithm III-E-2, Case 2.

Hence at the end of the (e + 1)*th* iteration, $H_{ij}(T_{ij}) \equiv h_{ij}(T_{ij})$, using the induction hypothesis.

Case 2: $h_{ij}(NEWT_{ij}) = (h_{ij}(T_{ij}), (NEWT_{ij}, NEWM_{ij}))$.

Then $(NEWT_{ij}, NEWM_{ij})$ is output along line (i, j) and T_{ij} is updated to $NEWT_{ij}$ following the I/O operation. This follows directly from algorithm III-E-2, Case 1.

Hence at the end of (e + 1)*th* iteration, $H_{ij}(T_{ij}) \equiv h_{ij}(T_{ij})$.

Case 3: $h_{ij}(NEWT_{ij})$ is none of the above.

Then there exists (t', m'), $t' < NEWT_{ij}$, in $h_{ij}(NEWT_{ij})$. This violates the operation of algorithm III-E-2, since $NEWT_{ij}$ is set to the smallest t for which a (t, m) is in the history. Hence this case is impossible!

Thus we can assert following the (e + 1)*th* iteration that,

$$H_{ij}(T_{ij}) \equiv h_{ij}(T_{ij}).$$

Applying induction on e, the theorem is thus proven for all e.

2) Correctness of System Output

Theorem 3 (correctness of process output) is not sufficient by itself for system correctness since there may appear to be a possibility that all outputs in a loop could be incorrect. We, however, show that this cannot happen; i.e., at any point in simulation all sequences transmitted along all lines are correct.

Simulation proceeds by LP's executing their I/O operations in an asynchronous fashion. At any point in simulation, many LP's may be in the process of executing their I/O operations simultaneously. We define an LP i to be in a halt *state* if $T_i \geqslant Z$; i.e., no computation or I/O operation takes place in LP i. LP i is in a *wait* state if it is waiting for some I/O operation. LP's i and j *are waiting for each other* if LP i is waiting to output to LP j and LP j is waiting to input from LP i.

In a simulation run, the only *events* that we consider are the transmissions of tuples. For any particular simulation run, we may order the events in the sequence in which they occur in that run and if more than one event happens simultaneously, we order the simultaneous events arbitrarily; note that for different simulation runs different sequences of events may result. The actual instants at which the tuples are transferred are of no consequence; only the above ordering among events is of importance.

We formally define a simulation run R by a sequence of four tuples:

$$R = ((t_1, m_1, i_1, j_1) \cdots (t_k, m_k, i_k, j_k) \cdots (t_r, m_r, i_r, j_r)) \tag{23}$$

(t_k, m_k, i_k, j_k) denotes that the k*th* event to happen in the logical system (in the above ordering) is the transfer of the tuple (t_k, m_k) along line (i_k, j_k). Following the r*th* event, all LP's are in either halt or wait state and no further event is possible; i.e., no two LP's are waiting for each other. Note that t_{k+1} may be smaller than t_k (provided $(i_k, j_k) \neq (i_{k+1}, j_{k+1})$); this means that the logical system *does not* duplicate the sequential behavior of the physical system; a snapshot of the logical system taken after the k*th* event may correspond to no physical time instant since clock values of all lines will not in general be identical.

Let R_e denote the initial subsequence of R consisting of the first e 4-tuples. Define R_e to be *correct* if and only if for every 4-tuple (t_k, m_k, i_k, j_k), $k \leqslant e$:

$$H_{i_kj_k}(t_k) \equiv h_{i_kj_k}(t_k) \tag{24}$$

A *simulation run R is correct* if R_e is correct for all e and there exists (Z, m, i, j) in R, for every line (i, j). The *logical system is correct* if all possible simulation runs are correct.

(25)

Theorem 4: (Correctness of System Output)

For every simulation run R, R_e is correct for all e.

Proof: For any simulation run R, we show that R_e is correct for $e = 0, 1, \cdots r$. Proof is by induction on e. For $e = 0$, $R_e = (\ )$ and hence the theorem holds trivially. Now assume for any $e \geqslant 1$ that R_{e-1} is correct. Then it is sufficient to show for correctness of R_e that,

$$H_{i_ej_e}(t_e) \equiv h_{i_ej_e}(t_e)$$

Consider the LP i_e following the transfer of the tuple (t_e, m_e) from i_e to j_e. The history of outputs on line (i_e, j_e) up to and including the tuple (t_e, m_e), i.e., $H_{i_ej_e}(t_e)$, is a function of the inputs received so far by i_e, i.e., inputs that appear in R_{e-1}. According to the induction hypothesis, these inputs are all correct and applying Theorem 3 the corresponding output on line (i_e, j_e), i.e., $H_{i_ej_e}(t_e)$, is correct. Hence,

$$H_{i_ej_e}(t_e) \equiv h_{i_ej_e}(t_e).$$

C. Absence of Deadlock

We show in this section that the logical system cannot stop prematurely; i.e., no simulation run can stop (a simulation stops when every process is in a halt or wait state and no two processes are waiting for each other) unless every line's clock value reaches Z. Another fact that is proven in the following subsection is that every line's clock value reaches Z in a finite number of simulation steps in every simulation run.

Call a line (i, j) WN line (waiting-nonwaiting) at some point in the simulation if LP i is waiting to output LP j, LP j is not waiting for input from LP i, and both LP's i, j are in a wait state. This means that LP j is waiting for some other LP K, $K \neq i$. Similarly line (i, j) is NW (nonwaiting-waiting) at some point in the simulation if LP i is not waiting to output to LP j, LP j is waiting for input from LP i, and both LP's are in a wait state. NN and WW lines may be defined similarly.

1) Clock Values of Waiting Processes

Theorem 5: (Clock Values of Waiting Processes are Higher)

At any point in the simulation,

1) if (i, j) is a WN line then $T_i > T_j$, and
2) if (i, j) is a NW line then $T_i \leqslant T_j$, and
3) if (i, j) is NW line and is predictable then $T_i < T_j$.

Proof: We observe the following for any LP i which is in a wait state.

a) If i is waiting on some line then the clock value of LP i = clock value of this line.

b) If LP i is not waiting on some input line (k, i) then,

$$T_i < T_{ki}.$$

This follows from definition, if $k \notin NEXT_i$. If $k \in NEXT_i$, then i must have already read the input along (k, i) and is now waiting on some other line. Prior to input along (k, i), $T_{ki} = T_i$. Following input, T_{ki} has increased (system chronology theorem) and T_i has remained the same, since it is still in a wait state. Thus the result follows in either case.

c) If LP i is not waiting for an output line (i, j) then $T_i \leqslant T_{ij}$; using arguments similar to b). $T_i = T_{ij}$ is possible when $j \notin NEXT_i$, if $TOUT_{ij} = T_{ij}$.

We now use these observations in proving the theorem.

1) (i, j) is a WN line.
From observation (a), $T_i = T_{ij}$.
From observation (b), $T_j < T_{ij}$.
Hence the result.

2) (i, j) is an NW line.
From observation (c), $T_i \leqslant T_{ij}$.
From observation (a), $T_j = T_{ij}$.
Hence $T_i \leqslant T_j$.

3) If (i, j) is predictable and an NW line then,

$$TOUT_{ij} \geqslant T_i + L_{ij}(T_i) > T_i$$

Since LP i is not waiting on (i, j), then either $j \notin NEXT_i$, i.e., $T_i < T_{ij}$, or $j \in NEXT_i$ and output has already been performed along (i, j) and hence $T_i < T_{ij}$. Since $T_j = T_{ij}$ from observation a), the result follows.

2) Absence of System Deadlock

Define the logical system to be *deadlocked*, at some point in the simulation if,

1) there exist one or more LP's in a wait state, and

2) there are no two LP's waiting for each other; if 2 LP's are waiting for each other a message will be sent from one LP to the other and the simulation is not yet over.

Theorem 6: (Absence of System Deadlock)

The logical system is never deadlocked in any simulation run.

Proof: Assume the converse. Then there is an LP i which is in a wait state. If LP i is waiting on line (i, j) (or j, i) then LP j cannot be in a halt state since one of its incident lines has $T_{ij}(T_{ji}) < Z$. Hence for every LP i which is in a wait state there exists j such that either (i, j) is a WN line or (j, i) is an NW line (and j is in a wait state).

Hence, construct a sequence of LP's $i_0, i_1 \cdots$ where i_0 is some LP in wait state and for every q, either (i_q, i_{q+1}) is a WN line or (i_{q+1}, i_q) is an NW line (and i_{q+1} is in a wait state). Since the number of LP's is finite, there exist K, L such that $i_K = i_L$. Let K, L be so chosen that all LP's $i_{K+1} \cdots i_L$ are distinct.

From Theorem 5, $T_0 \geqslant T_1 \geqslant \cdots T_K \geqslant \cdots \geqslant T_L \geqslant \cdots$. Since $T_K = T_L$, $T_K = T_{K+1} = \cdots = T_L$. Call this common clock value of processes T. If for some q, $K \leqslant q < L$, (i_q, i_{q+1}) is a WN line then $T_q > T_{q+1}$ (from Theorem 5) contradicting that $T_q = T_{q+1}$. Hence for every q, $K \leqslant q < L$, (i_{q+1}, i_q) is an NW line. Thus the sequence $i_L, \cdots, i_{q+1}, i_q \cdots i_K$ forms a directed cycle. From the predictability property (Section II-D) there exists a predictable line (i_{q+1}, i_q) in this cycle. Since (i_{q+1}, i_q) is an NW line and predictable, it follows from Theorem 5 that $T_{i_q} > T_{i_{q+1}}$ contradicting that $T_{i_q} = T_{i_{q+1}}$.

Corollary: At the end of the simulation, every LP is in a halt state.

D. Termination

We show that after a finite number of steps, every simulation run stops, i.e., every LP enters a halt state. Note that absence of deadlock (Section VII-C) implies that there will be no premature stopping; it does not rule out the possibility of an infinite simulation where the clock value of some line never gets up to Z.

1) Bounded Properties of Lines Incident on a Process

Theorem 7: Let $\tau < Z$ be the clock value of any given predictable line (i, j) at some point in the simulation. Then no more than $2\tau/\epsilon$ tuples could have been transmitted along that line up to that point in the simulation.

Proof: From (2) and the system correctness theorem (Theorem 4), there can be no more than τ/ϵ tuples with non-NULL m-values transmitted along any line whose clock value is τ.

We next show that there can be no more than τ/ϵ tuples with NULL m-values transmitted along any predictable line with clock value τ. (Note that no tuple with t-value 0 is actually sent.) This is shown by proving the clock value of a predictable line increases by at least ϵ whenever a tuple with a NULL m-value is output. More precisely, we will show that, following the computation step in the algorithm of Section III-F, if

$$\mathrm{NEWT}_{ij} < Z \text{ and } \mathrm{NEWM}_{ij} = \mathrm{NULL} \text{ then } \mathrm{NEWT}_{ij} \geqslant T_{ij} + \epsilon. \tag{26}$$

Since $\mathrm{NEWT}_{ij} < Z$ and $\mathrm{NEWM}_{ij} = \mathrm{NULL}$, it follows from algorithm III-E-2 that

$$\mathrm{NEWT}_{ij} = \mathrm{TOUT}_{ij}. \tag{27}$$

$$\begin{aligned}\mathrm{TOUT}_{ij} &= \mathrm{TIN}_i + L_{ij}(\mathrm{TIN}_i) \ [(18)]\\ &\geqslant T_{ij} + L_{ij}(T_{ij}), \text{ this follows from } \mathrm{TIN}_i \geqslant T_{ij}\\ &\quad \text{since } j \in \mathrm{NEXT}_i\\ &\geqslant T_{ij} + \epsilon, \text{ if line } (i, j) \text{ is predictable.}\end{aligned} \tag{28}$$

Equation (26) follows from (27) and (28). Hence in any interval $\tau < Z$, there cannot be more than τ/ϵ tuples with NULL m-values transmitted along a predictable line (i, j).

Definition: A sequence of tuples along any line is *linearly bounded* if there exist constants a, b such that for every τ, $0 \leqslant \tau < Z$, there are no more than $a\tau + b$ tuples in the sequence each of whose t-values is smaller than τ.

Note that Theorem 7 implies that the sequence of tuples on any predictable line is linearly bounded.

Theorem 8: For any LP i, if all of its input sequences are linearly bounded then all of its output sequences are linearly bounded.

Proof: Let the LP i have I input lines indexed $1, 2 \cdots I$. Since each input sequence is linearly bounded, for the *kth* input line there exist constants a_k, b_k such that no more than $a_k\tau + b_k$ tuples with t-values smaller than τ are received along that line. Hence the total number of tuples with t-values smaller than τ, received by this LP, do not exceed $A\tau + B$, where

$$A = \sum_{k=1}^{I} a_k \qquad B = \sum_{k=1}^{I} b_k.$$

For any output line (i, j), since TOUT_{ij} is exclusively a function of input histories, it follows that TOUT_{ij} cannot change unless TIN_i changes. Once a tuple $(\mathrm{TOUT}_{ij}, \mathrm{NULL})$ has been output, no more tuples can be output along the line (i, j) unless TOUT_{ij} changes (see algorithm III-F), i.e., unless TIN_i changes. Furthermore, every tuple with a NULL m-value is of type $(\mathrm{TOUT}_{ij}, \mathrm{NULL})$. TIN_i changes only when messages are received and this cannot happen more than $A\tau + B$ times and hence there cannot be more than $A\tau + B$ NULL tuples on any output line.

From (2) and the system correctness theorem (Theorem 4) there can be no more than τ/ϵ tuples with non-NULL m-values on any line (i, j). Hence the total number of tuples along line (i, j) cannot exceed $(A + 1/\epsilon)\,\tau + B$. This proves the theorem. □

2) Bounded Property of the System

Theorem 9: The sequence of tuples on all lines are linearly bounded.

Proof: Proof is by contradiction. Consider any line (i, j) whose tuple sequence is not linearly bounded. From Theorem 8 it follows that there exists an input line (k, i) to i whose tuple sequence is not linearly bounded. Continuing in this manner, we must either reach a source (contradiction) or form a loop. Since every loop has at least one predictable line, this loop must have at least one line with a linearly bounded sequence (Theorem 7). Contradiction!

E. System Correctness

Recall that the system is defined to be correct if every simulation run is correct and every simulation run ends with clock value of every line at Z. We have shown correctness of system output (Theorem 4), absence of deadlock (Theorem 6), and a bounded property of sequences (Theorem 9) for every simulation run. Combining these results we prove system correctness.

Theorem 10: (Termination)

The clock value of every line reaches Z in a finite number of steps in any simulation run.

Proof: Theorem 9 implies that the sequence of tuples on every line is linearly bounded. Hence, there exist constants A and B such that the sequence for any line can have no more than AZ + B tuples. Thus r, the length of the simulation run cannot exceed (AZ + B) × (number of lines in the system). Hence every simulation terminates (in a finite number of tuple transmissions). From the corollary to Theorem 6 (absence of system deadlock), it follows that no simulation run can terminate with the clock value of any line smaller than Z. This proves the theorem.

Theorem 11: The logical system is correct.

Proof: Combining Theorem 4 and Theorem 10, the result follows.

Appendix
Encapsulating the Physical Process

We have noted in the Introduction that we observe two distinct aspects in the simulation of a physical system.

1) Simulation of individual physical processes (which corresponds to computing (t, m) in algorithm III-E-2) which effectively simulates the computation of $F_{ij}(\ )$ as defined in (8), for every (i, j).

2) Selection of lines for input and output operations and the actual I/O operations which have been described in algorithm III-F and whose correctness has been demonstrated in Section VI.

The outcome of such a separation is that design and correctness proofs of 1) and 2) may be carried out independently.

We now discuss methods for simulating individual physical processes.

Definitions: Recall that s_{ij} is the tuple sequence of arc (i, j) in the physical system [see (3)]. Let

$$s_{ij} = ((t_1, m_1), \cdots, (t_K, m_K)) \tag{A1}$$

Define $\mathrm{INFORM}_{ij}(t)$, the *information on physical arc* (i, j) *at time* t, as the message (if any) being transmitted on that arc at t, and as NULL if there is no message on arc (i, j) at t.

$$\mathrm{INFORM}_{ij}(t) = \begin{cases} \mathrm{NULL} & \text{if } t \neq t_q, q = 1, \cdots, K \\ m_q & \text{if } t = t_q \end{cases} \tag{A2}$$

Define $h_{ij}(t', t'')$, for $t'' > t'$, *the history of messages on arc* (i, j) *in the interval* $(t', t'']$, as the tuple sequence on arc (i, j) in that interval. Formally, $h_{ij}(t', t'')$, with $t'' > t'$ is that tuple sequence, such that

$$h_{ij}(t'), h_{ij}(t', t'') = h_{ij}(t'') \tag{A3}$$

where "," represents the appending of two tuple sequences. We define $h_{ij}(t, t) = \mathrm{INFORM}_{ij}(t, t)$. Let $\mathrm{STATE}_i(t)$ be the state of the physical process at time t; we assume that the state at t+, the instant after t, depends only on the state at t and the information received by the process at t. Similarly, the output from process i at t depends on the state of process i and the information received by process i at t. Let

$$\mathrm{STATE}_i(t+) = b_i(\mathrm{STATE}_i(t), \mathrm{INFORM}_{1i}(t), \cdots, \mathrm{INFORM}_{Ni}(t)) \tag{A4}$$

and

$$\mathrm{INFORM}_{ij}(t) = c_{ij}(\mathrm{STATE}_i(t), \mathrm{INFORM}_{1i}(t), \cdots, \mathrm{INFORM}_{Ni}(t)) \tag{A5}$$

Note that the number of states may be potentially infinite. We also assume the existence of computable functions D_{ij} and C_{ij} such that

$$L_{ij}(t) = D_{ij}(\mathrm{STATE}_i(t), \mathrm{INFORM}_{1i}(t), \cdots, \mathrm{INFORM}_{Ni}(t)) \tag{A6}$$

$$h_{ij}(t, t + L_{ij}(t)) = C_{ij}(\mathrm{STATE}_i(t), \mathrm{INFORM}_{1i}(t), \cdots, \mathrm{INFORM}_{Ni}(t)) \tag{A7}$$

In other words the lookahead value at any time on any arc (i, j), and the message history on that arc between the current time and the point to which the process can look ahead, is computable from the state of process i at the current time and the information received at this time.

For simulating the physical system we assume the existence of a computable function $B_i(\ )$, such that for any two times t′ and t′, where $t' < t''$,

$$\mathrm{STATE}_i(t'') = B_i(t'', \mathrm{STATE}_i(t'), h_{1i}(t', t''), \cdots, h_{Ni}(t', t'')) \tag{A8}$$

In other words the state of a physical process at time t″ can be computed from the state of that process at an earlier time t′ and the messages which arrive between t′ and t″.

$B_i(\ )$ may appear to be a complex function in comparison with b_i because its arguments include histories. However, in practice, computation of the next state using B_i is simple because we will only use histories which are either ( ) or consist of a single tuple (t″, m), in (A8).

Note that B_i can be computed in the logical system by actually running the physical process (which is equivalent to obtaining an evaluation of b_i and c_{ij} at all points), in the interval $(t', t'']$; of course, in practice other techniques are preferred. B_i and C_{ij} simulate b_i and c_{ij}, respectively.

We now expand on the iteration cycle of algorithm III-F, for process i. The algorithm proceeds by repeatedly evaluating functions B, C, and D.

Selection Step

The logical process has saved $\mathrm{STATE}_i(T_i)$ and (T_{ki}, M_{ki}) for each input line k, i. Let T'_{ki} be the last value of T_{ki}.

$$\mathrm{INFORM}_{k,i}(T_i) = \begin{cases} \mathrm{NULL} & \text{if } T_{ki} > T_i > T'_{ki} \\ M_{ki} & \text{if } T_{ki} = T_i \end{cases}$$

Compute $L_{ij}(T_i)$ from (A6), and then NEXT_i as given in algorithm III-F.

For each output line (i, j) such that j is in NEXT_i, we compute $h_{ij}(T_i, T_i + L_{ij}(T_i))$ from (A7). We next compute (NEWT_{ij}, NEWM_{ij}) the next tuple to be output on line (i, j) which is defined as:

$$(\mathrm{NEWT}_{ij}, \mathrm{NEWM}_{ij}) = \begin{cases} (T_i + L_{ij}(T_i), \mathrm{NULL}), \\ \quad \text{if } h_{ij}(T_i, T_i + L_{ij}(T_i)) \text{ is empty} \\ \text{first tuple in} \\ \quad h_{ij}(T_i, T_i + L_{ij}(T_i)) \text{ otherwise} \end{cases}$$

After the I/O Operation

Compute NEWT_i, the new value of T_i. Obtain $h_{k,i}(T_i, \mathrm{NEWT}_i)$ as follows:

$$h_{ki}(T_i, \mathrm{NEWT}_i) = \begin{cases} (\,) & \text{if } T_{ki} > \mathrm{NEWT}_i \\ (T_{ki}, M_{ki}) & \text{if } T_{ki} = \mathrm{NEWT}_i \end{cases}$$

Compute $\mathrm{STATE}_i(\mathrm{NEWT}_i)$ from (A8). Set $T_i = \mathrm{NEWT}_i$.

The amount of memory required by logical process i is the amount required to store the state information (which is also required in the physical process) and for the local variables T_i, T_{ki}, M_{ki}, T_{ij}, M_{ij}, NEWT_{ij}, NEWM_{ij} and the amounts required to evaluate functions B_i, C_{ij}, and D_{ij}; we assume that the resources (and the times) required to evaluate these functions are bounded. The assumption about requiring bounded resources is not restrictive because if unbounded resources were required no computing system could simulate the physical process.

The simplicity of updating states and output information is achieved by a judicious design of the algorithm, in particular by the definition of NEXT.

Acknowledgment

We are grateful to the University of Waterloo students who provided constructive criticism during a course on this topic in the summer of 1977 and especially to Prof. E. Manning, Prof. J. Wong, and Prof. K. Peacock at the University of Waterloo. Discussions with Prof. C. A. R. Hoare on communicating sequential processes led ,to considerable simplifications in our design. We are indebted to Prof. J. Dennis who brought the work of R. E. Bryant to our attention. We are grateful to Dr. E. W. Dijkstra and Dr. H. D. Mills for comments on earlier drafts. D. Davis has done an excellent job of typing under adverse conditions.

References

[1] K. M. Chandy, V. Holmes, and J. Misra, "Distributed simulation of networks," Dep. Comput. Sci., Univ. Texas at Austin, TX, Tech. Rep. 81; also, *Computer Networks*, to be published.

[2] K. M. Chandy and J. Misra, "A non-trivial example of concurrent processing: Distributed simulation," Dep. Comput. Sci., Univ Texas at Austin, TX, Tech. Rep. 82; also, in *Proc. COMPSAC*, Chicago, Nov. 16–18, 1978.

[3] J. B. Dennis, "First version of a data flow procedure language," MAC Tech. Memo 61, MIT, Nov. 1973.

[4] ——, "Packet communication architecture," in *Proc. 1975 Sagamore Comput. Conf. Parallel Processing*, Aug. 1975, pp. 224–229.

[5] E. W. Dijkstra, "Notes on structured programming," in *Structured Programming*, C. A. R. Hoare, Ed. New York: Academic, 1972.

[6] E. W. Dijkstra *et al.*, "On-the-fly garbage collection: An exercise in cooperation," Working Material NATO Summer School on Language Hierarchies and Interfaces, Munich, 1975.

[7] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Commun. Ass. Comput. Mach.*, vol. 17, pp. 643–644, Nov. 1974.

[8] A. N. Habermann, "Synchronization of communicating processes," *Commun. Ass. Comput. Mach.*, vol. 15, pp. 171–176, 1972.

[9] C. A. R. Hoare, "Communicating sequential processes," *Commun. Ass. Comput. Mach.*, vol. 21, no. 8, Aug. 1978.

[10] C. A. R. Hoare and W. H. Kaubisch, "Discrete event simulation based on communicating sequential processes," unpublished.

[11] S. S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs," *Acta Informatica*, vol. 6, no. 3, pp. 319–340, 1976.

[12] S. S. Patil, "Coordination of asynchronous events," Ph.D. dissertation, MIT, May 1970.

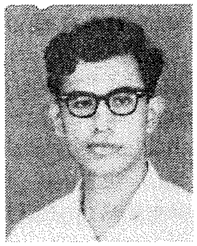
[13] M. Seethalakshmi, "Performance analysis of distributed simulation," Univ. Texas at Austin, M.S. Rep. in preparation.

[14] N. Wirth, "Program development by stepwise refinement," *Commun. Ass. Comput. Mach.*, vol. 14, pp. 221–227, 1971.

[15] V. Holmes, Ph.D. dissertation, Univ. Texas, 1978.

[16] J. D. Peacock, J. W. Wong, and E. Manning, "Distributed simulation using a network of microcomputers," *Computer Networks*, to be published.

[17] K. M. Chandy and J. Misra, Dep. Comput. Sci., Univ. Texas at Austin, TX, Tech. Rep. TR-86.



K. Mani Chandy received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Madras, India, the M.S. degree in electrical engineering from the Polytechnic Institute of Brooklyn, Brooklyn, NY, and the Ph.D. degree in operations research from the Massachusetts Institute of Technology, Cambridge, in 1965, 1966, and 1969, respectively.

From 1966 to 1967 he was an Associate Engineer with Honeywell EDP and from 1969 to 1970 he worked as a Staff Member at the IBM Cambridge Scientific Research Center. He has also been a consultant to the Computer Sciences Department, Thomas J. Watson Research Center. He is presently Professor of Computer Sciences and Electrical Engineering, University of Texas, Austin, and a consultant for Information Research Associates, Inc., Austin, TX. His current research interests include modeling of computer systems, networks, and reliability.

Dr. Chandy is a member of the Association for Computing Machinery.

Jayadev Misra (S'71–M'72) received the B. Tech. degree in electrical engineering from the Indian Institute of Technology, Kanpur, in 1969, and the Ph.D. degree in computer science from The Johns Hopkins University, Baltimore, MD, in 1972.

He worked for IBM, Federal System Division, from January 1973 to August 1974. He is currently an Associate Professor with the Department of Computer Sciences, University of Texas, Austin.

Dr. Misra is a member of the Association for Computing Machinery.