MIT/LCS/TR-188

# SIMULATION OF PACKET COMMUNICATION ARCHITECTURE COMPUTER SYSTEMS

Randal E. Bryant

MIT/LCS/TR-188

# SIMULATION OF PACKET COMMUNICATION ARCHITECTURE COMPUTER SYSTEMS

by

Randal Everitt Bryant

November, 1977

This research was conducted under a graduate fellowship from the National Science Foundation. Additional funding was supplied by the National Science Foundation under grant DCR75-04060, and by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract no. N00014-75-C-0661.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LABORATORY FOR COMPUTER SCIENCE
(formerly Project MAC)

CAMBRIDGE MASSACHUSETTS

# SIMULATION OF PACKET COMMUNICATION ARCHITECTURE COMPUTER SYSTEMS[a]

by

Randal Everitt Bryant

## ABSTRACT

Simulations of computer systems have traditionally been performed on a single, sequential computer, even if the system to be simulated contains a number of components which operate concurrently. An alternative would be to simulate these systems on a network of processors. With this approach, each processor would simulate one component of the system, hence the component simulations could proceed concurrently. By exploiting the modularity and concurrency in the system to be simulated, the simulation would itself be modular and concurrent.

An accurate simulation must model the time behavior of the system as well as its input-output behavior. In order to avoid real-time constraints on the processors and communication network in the simulation facility, the simulation of the timing must use a time-independent algorithm. That is, the simulated behavior of each component should not depend on the speed at which the simulation is performed.

With this time-independent approach, additional *coordination* operations are required to prevent a deadlock of the simulation. This coordination can be provided without any centralized control. Instead, the program for the simulation of each component is modified, so that each component simulation will communicate status information to other component simulations. Additional *termination* operations are also required to assure that the simulation will terminate under the exact same conditions that the system being simulated would terminate. These operations can also be provided without any centralization of control or real-time constraints. Furthermore a simulation which uses these coordination and termination operations is provably correct. That is, the simulation will accurately model both the time behavior and the input-output behavior of the system.

THESIS SUPERVISOR: Jack B. Dennis
TITLE: Professor of Electrical Engineering and Computer Science

[a]This report is based upon a thesis of the same title submitted to the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology on May 30, 1977 in partial fulfillment of the requirements for the degree of Master of Science.

## Acknowledgements

I would like to thank the members of the Computation Structures Group at the MIT Laboratory for Computer Science, especially Professor Jack B. Dennis and Ken Weng, for suggesting this area of research and for providing valuable feedback during the research and writing processes. I would also like to thank the National Science Foundation for providing me with financial support during my studies through their graduate fellowship program.

## Table of Contents

# Chapter 1

# Introduction

Computer Systems have traditionally been simulated on a single, sequential computer, even if the system to be simulated contains a number of components which operate in parallel. One of the primary purposes of simulation languages, such as GPSS and Simscript II [13], is to order the simulation of the events occuring in the different components in such a way that the simulation will correctly model the operation of the system to be simulated. An alternative approach would be to simulate parallel systems on a network of computers, such as a network of microprocessors [2,14,21] or the Arpanet [15], where each processor would simulate the operations of one component of the system. This would allow the simulation to exploit the modularity and concurrency of the system to be simulated and thereby itself achieve a high level of modularity and concurrency. The simulation of packet communication architecture systems [6] seems particularly suited for this approach, since these systems are highly modular - the components of the system operate independently and communicate with each other only by sending message packets. Hence these systems can be simulated by a network of processors which communicate by message passing.

## Packet Communication Architecture

A packet communication architecture system consists of a number of independent processor modules which communicate by sending packets of information to one another. A single program is implemented as a number of separate processes, such that each process runs on one of the modules, hence the components of the program can be executed in parallel.

The modules in a packet communication architecture system can communicate only in a limited fashion. All communication with a module is in the form of packets, except the initial state of the module, which can be given to the module in nonpacket form. Thus, a module could be initialized with a program and initial data, but thereafter it can receive information only in packets. Furthermore, a module can communicate with only a limited number of other modules. Each module receives and sends out packets through its input and output ports. A particular input port to a module can receive packets only from a particular output port of some module, or from a particular source outside the system. Input ports of the latter type are called *system input ports*, since they are the only means for an external source to send data to the system. Similarly, from a particular output port of a module, packets can be sent only to a particular input port of some module or to a particular external destination. Output ports from which packets are sent to external destinations are called *system output ports*.

Packets are carried along one-way data channels from the output port of one module to the input port of another. These channels cannot alter the values of the packets, and they must preserve the sequential ordering of the packets. Thus, a channel can be viewed as a FIFO queue between two ports. The interconnections between modules cannot be changed dynamically.

The modules in a packet communication architecture system operate autonomously. There is no central control in the system, and any monitoring of the system operation must be *passive*. That is, only an external observer is allowed to monitor the modules or channels in the system, and the monitoring is not vital to the system's correct operation. As a result of this autonimity, a module can operate as soon as the necessary data packets have arrived regardless of the status of other modules in the system.

A packet communication architecture system is designed so no component of the system will be required to fulfill any timing constraints. Instead, the system must be designed to operate correctly regardless of the delay times or throughputs of the modules and channels. For example, one module cannot require another module to have a minimum response time. As a result, modules must use asynchronous communication protocols, so that a module cannot send a data value to another module which lacks sufficient buffer space. This communication protocol, however, must be implemented as packets sent back and forth between two modules for each data transfer. Otherwise, an acknowledgement signal received from a module to which data has been sent would constitute a form of nonpacket input information.

As a consequence of this time-independent design, the speed of the system or any of its components is a performance issue and not a necessary requirement for correct operation. If one module or channel is particularly slow, it might slow down the entire system, but it will not cause any malfunctions.

Examples of packet communication architecture systems include the data flow processors of Dennis and Misunas [7,8,9] and the data flow processor of Rumbaugh [20]. While not precisely a packet communication architecture system (due to dynamically changeable interconnections) the Distributed Computing System at the University of California, Irvine, when running with the DCOS operating system [19], embodies many of the same design philosophies.

## Advantages of Packet Communication Architecture Systems

These systems have several major advantages over both traditional computer systems and other designs for parallel systems. First, the modules in the system can operate concurrently, thereby achieving a high rate of computation. Since there is no central control, there is no component which will inherently cause a bottleneck in the system, or which must have an extremely high throughput in order to keep the rest of the system operating at a reasonable rate.

Second, the system can be designed modularly, by first specifying the functional requirement for each module as well as some connection standards and then designing the modules individually. Since modules can interact only in limited and well-defined ways, as opposed to systems which contain shared memories or allow interrupts, for example, a module has a very clean interface with the rest of the system. Furthermore, since there are no timing restrictions on a module, the specifications for its operation need contain only its functional operation, i.e. the output packets produced in response to a set of input packets.

Once a system has been designed, we can try to maximize its performance. This involves identifying the modules and channels in the system which are consistently heavily loaded and hence form bottlenecks in the system. A bottleneck can be eliminated by redesigning the module or channel to operate faster or by splitting one module into several modules. Because the system is designed to be speed independent, the speed of one module can be varied without causing malfunctions.

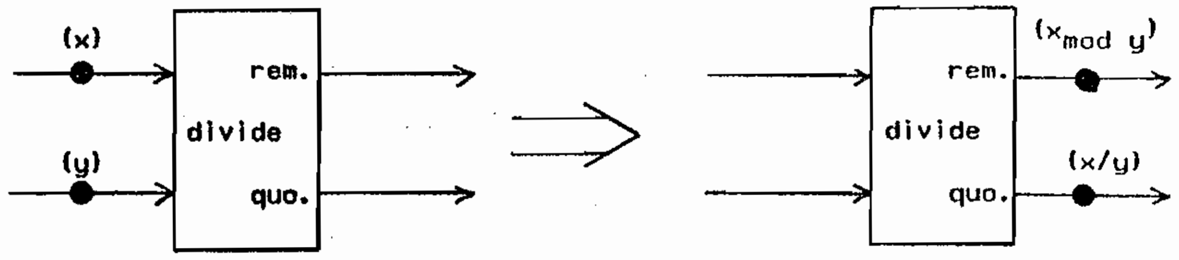
One further result of this modularity of design is that these systems can be proved correct much more easily than other computer systems. To prove the correctness of a packet communication architecture system, one can specify the required properties of each module, prove that each module satisfies these properties, and then prove that the system will operate correctly if all modules satisfy their requirements. In other words, the correctness of the system can be proved modularly. General methods of proving the correctness of packet communication architecture systems are currently being investigated by Ellis [10].

## Examples of Packet Communication Architecture Modules

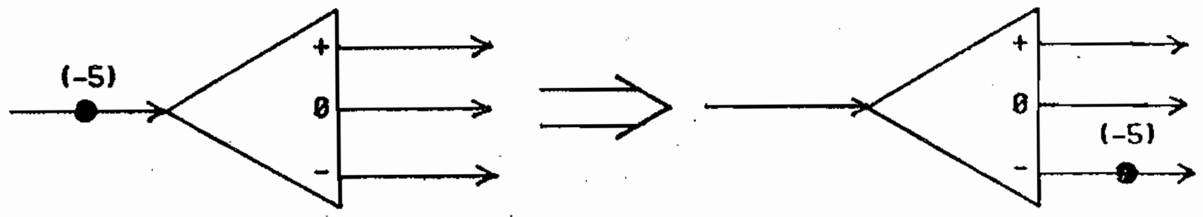
Three basic module types: functional operators, switches, and arbiters illustrate some of the operations which can be performed by packet communication architecture modules. Examples of their operation are shown in Figure 1.1. In the diagrams the lines represent the channels connected to the input and output ports of the modules, and the dots on these lines represent data packets being transmitted over the channels.

A functional operator computes several functions (one for each output port)

A. Functional Operator



B. Switch



C. Arbiter

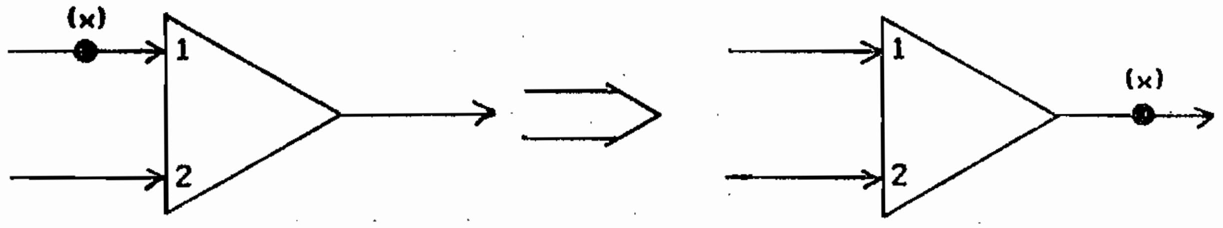


*Figure 1.1* - **Examples of Operation for Three Simple Module Types.**

with input packets as arguments. It can *fire* as soon as one packet is received at each input port, meaning that it absorbs these input packets, computes the output values, and sends one output packet from each output port. For example, the DIVIDE module of Figure 1.1a computes two functions: the quotient and the remainder of the input values.

A switch module provides a means of routing data to different modules in the system. It can fire as soon as a packet is received on its input port. In firing, it absorbs the input packet and then sends an identical output packet from one of several output ports, depending on the packet's value. In the example of Figure 1.1b, the output port selected depends on whether the packet value is positive, zero, or negative.

As a final example, the arbiter module serves to merge together the streams of output packets from several modules. It can fire as soon as a packet is received on either input port. In firing, it absorbs a packet from one of the input ports and sends an identical packet from its output port. If packets are received at two input ports simultaneously, the module will first fire, absorb one of these packets, and send it out. By the rules of operation, any packet which is not absorbed will remain at the input port. Hence, the module will fire a second time, absorb the remaining packet, and send this one out.

Other packet communication architecture modules can have behaviors which depend on other factors, such as past activities of the module, the arrival times of the input packets, and stochastic processes within the module. The general rules of operation for the modules will be discussed in Chapter 2.

## The Need for Simulation

Once the functional behavior of all components have been developed and proved correct, there are other issues to be settled before the system can be implemented. The implementation must meet other requirements on the overall speed of operations or the total cost of the system. Thus, for a particular implementation, a designer will want to measure the performance of the system for different sets of input data. These measurements can include such factors as the overall speed of the system, the load on particular components, and the buffering requirements at the input ports. Once measurements for a particular implementation have been made, the designer will want to make measurements when such parameters as throughput or delay time for particular components have been varied, or modifications have been made to the original design. By this method, the designer can maximize the speed and minimize the cost of the system.

Measurements of a system's performance are required not only to find an optimum implementation, but also to compare the system to other system designs, or to conventional computer systems. While packet communication architecture systems are potentially very fast due to the high level of parallelism, a method of comparison with traditional computer systems is desired.

Developing mathematical methods of predicting the performance of particular systems seems to be very difficult. One cannot simply count the number of instruction cycles required for a particular program with a particular set of input data. While the modules interact with each other in a very limited and well-defined way from a functionality viewpoint, the performance of a module can have very subtle effects on the performance of the overall system. For example, increasing the throughput of one module can cause another module to become a bottleneck in the system. Thus, a "modular" approach to performance analysis will not work. Furthermore, the system designer wants to know more than just the average or worst case performance of some system. He wants to know the detailed performance measurements for each component of the system. This amount of detail could never be provided accurately by a mathematical analysis of performance.

An accurate simulation of a system would provide the desired measurements for a particular set of input data. While it might be hard to judge the general performance of a system based on simulations for a few sets of input data, this approach seems to provide a great deal more information than analytic methods.

To avoid confusion between the system to be simulated and the system which performs the simulation, the former will be called the *actual* system, and the latter will be called the *simulation* system. Even though the "actual" system might in fact only exist on paper, this seems like a reasonable way to distinguish the two. Furthermore, the modules and channels of the actual system will be called the actual modules and actual channels.

## Requirements for the Simulation

To provide the type of measurements required to evaluate an implementation of a system, the simulation must accurately model all aspects of the system's operations. This includes modelling the detailed timing aspects of the system as well as the functional behavior. If only the functional aspects were modelled, the simulation would accurately model *some* implementation of the system, but most likely not the implementation we are interested in.

An accurate modelling of the system cannot rely on any stochastic methods of simulation, unless the modules themselves behave stochastically. For one thing, like analytic methods, methods of stochastically modelling packet communication architecture systems have not yet been developed. Thus, unless the system *is* affected by stochastic processes within the modules, a simulation of a system should provide all information about the activities of each module for a given set of initial states (i.e. module programs and initial data), and a particular sequence of input packets presented to each system input port. If the modules behave stochastically, the stochastic processes must be modelled, so that any random variables will be chosen with the same probability in the simulation as they are in the actual system. A single simulation will only model the system's activity for one choice of random variables, but a number of simulations can give an idea of the distribution of the system's performance.

## Methods of Simulation

One approach to the simulation of a packet communication architecture system is with a sequential computer system. With this approach, a single computer would simulate the activities of every module and every communication channel in the system. While this approach would be rather slow, it is not difficult to implement. For every packet on an input port of some module in the system, the simulation keeps a *packet descriptor* of the form $(M,p,v,t)$, where

> $M$ = the module number
> $p$ = the input port number
> $v$ = the value contained in the packet
> $t$ = the time at which the packet arrived at the input port.

These packet descriptors are stored as a sequential list called a *time line*, in which the descriptors are ordered by their time values. The simulation looks at the time line and decides which module in the system would fire the soonest. It then simulates the firing of this module by removing the absorbed input packets from the time line, computing the output values and delay time for the module, and then inserting new packet descriptors for each output packet into the time line. Each new packet descriptor contains the module and input port number of the input port which receives the packet, the value of this packet, and the time at which the input port would receive the packet. This process is repeated for the new time line, and so on, until no module in the system is able to fire. As long as the simulation always simulates the earliest firing in the system for a given state of the time line, it can be certain that all input packets which would have been received by this module at firing time are present on the time line. Since a module cannot be affected by new input packets arriving while it is firing, the entire firing of the module can be simulated without looking at other modules in the system. Simulation languages, such as GPSS and Simscript II [13], use a variant of this time line in simulating the activities of a number of concurrent processes on a single computer.

A large fraction of the simulation time will be spent looking at the time line to decide which module would fire earliest. Whereas it is not difficult to determine whether simple modules, such as functional operators, switches, or arbiters are ready to fire and at what time, these computations could take much longer for modules with more complex behavior. Moreover, as the size of the system increases, there will be more modules to check, and more descriptors on the time line. Hence, the time spent on overhead in the simulation can, in the worst case, increase as the square of the system size: there will be a linear increase in the total number of firings to be simulated, and for each firing a linear increase in the time required to decide which module would fire earliest. The time spent to actually simulate the activities of the modules, on the other hand, will increase only linearly with the system size. As the size of the system is increased, the proportion of simulation time spent on overhead will increase.

An alternative to simulation on a sequential computer is to simulate the system on a computer system consisting of a number of interconnected simulation processors, such as the Packet Architecture Simulation Facility of Leung, *et al* [14], shown in Figure 1.2. In this facility microprocessors serve as simulation processors. Each simulation processor simulates one or, for a large system, several of the modules in the system. The processors send packets to one another, just as the modules in the actual system would. The packets are sent over a communication network, which provides connections among all pairs of simulation processors. During a simulation, however, a processor would send packets to another processor only if the first is simulating a module which can send packets to a module being simulated by the second. The communication network is provided to allow the simulation of any system configuration. In addition, a host computer can load programs into the modules, initiate the simulation, and monitor its progress.

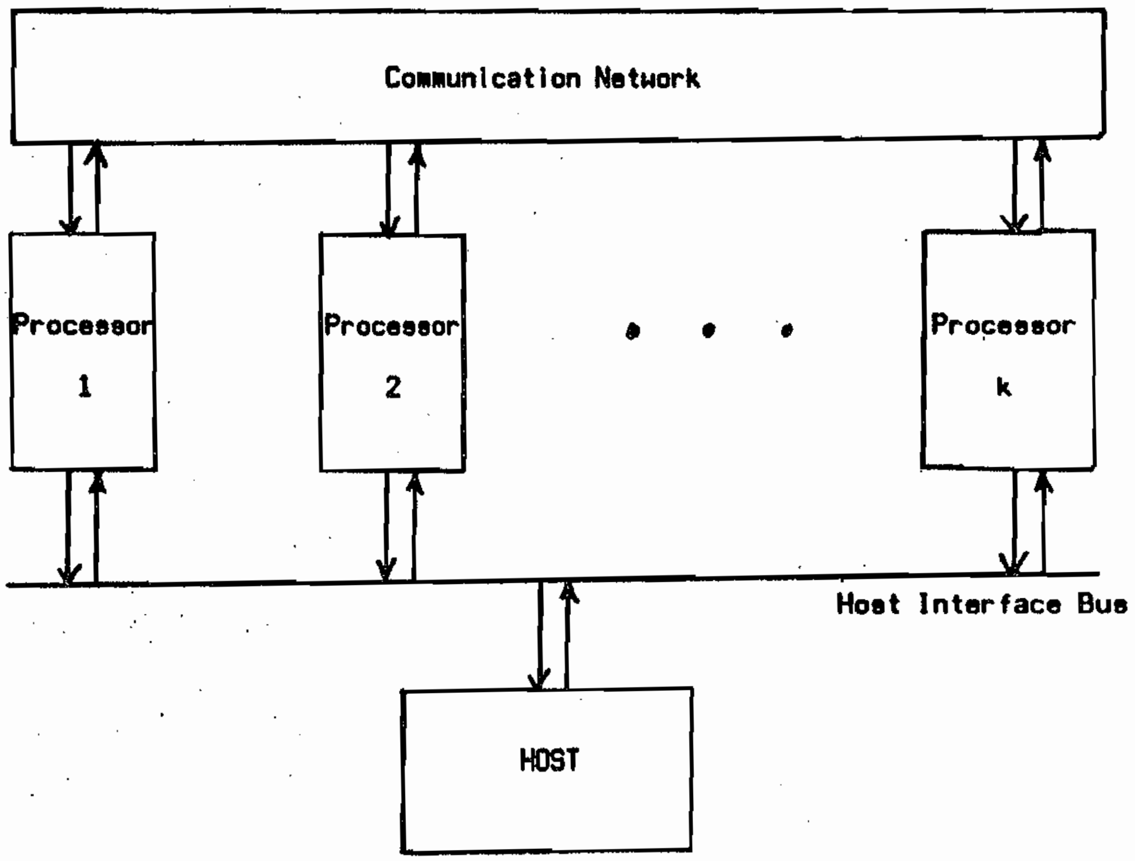


*Figure 1.2* - Structure of Simulation Facility

This approach seems very natural, since the structure of the simulation is much like that of the system being simulated. It should also be faster, since the simulation processors can operate in parallel. Hopefully, the amount of overhead will not be too great, either, so that a large fraction of processor time can be spent simulating the activities of the modules.

## Purpose of Thesis

In this thesis, methods of simulating packet communication architecture systems on a distributed computer system will be developed. The design goals for these simulation methods include:

> 1.) *Generality of Simulating System.* The simulation should not require a highly specialized computer system on which to perform the simulaton. It should work on any system which supports communicating processes, such as the Packet Architecture Simulation Facility [14], a network of microprocessors [2,21], the Distributed Computing System [11,19], or even more traditional systems such as the Burroughs B6700 [16].
>
> 2.) *Generality of Simulation.* The methods should enable the accurate simulation of any packet communication architecture system. A system designer should not be limited in the types of systems which he can simulate.
>
> 3). *Simplicity of Software.* The programs for each simulation processor should be reasonably simple to write, and short enough to be executed by small processors such as microprocessors.
>
> 4). *Reasonable Efficiency:* The simulation should make use of the potential parallelism in the simulation system. Furthermore, the amount of communication between processors to keep their efforts coordinated should be reasonably small.

One way to satisfy the first goal would be for the simulation itself to have the properties of a packet communication architecture system. First, the simulation processors should act autonomously, with no central control. This will simplify the computer system required to perform the simulation by removing the need for a highly specialized, high speed central controller. Of course, passive monitoring might be allowed to observe the simulation activities. Second, all communication between simulation processors should be in the form of packets. As a result, the processors will have a uniform form of input-output. Perhaps most importantly, the simulation will be time-independent. That is, the accuracy and correctness of the simulation will not depend on the speed of the simulation processors or the communication network. This will eliminate any real time constraints on the simulation hardware and software, which will greatly simplify the design. This will also enable the simulation to be performed on any computer system which supports communicating processes. The simulation of each component of a system could be handled by a different process. Several of these processes could be assigned to one processor, which could execute them without any time constraints.

While the simulation might be faster on a highly specialized simulation facility equipped with a high speed controller or processors designed for real time applications, the amount of time and money required to construct such a facility would be justified only if a very large number of simulations were to be performed.

The problem then becomes developing simulation methods based on packet communication architecture principles, which will satisfy the other three goals: generality, simplicity of software, and reasonable efficiency. One means of simplifying the task of software design is to take a modular approach to the design of simulation programs. The simulation program for a module must not only simulate the activities of the module, it must also communicate with other module programs to keep the simulation activities coordinated. Thus, the specifications for each simulation program will include not only specifications of the module to be simulated, but also specifications of the coordination activities. To keep the design modular, the coordination activities must be simple and uniform enough to be easily and accurately specified. Moreover, these coordination activities must be both general and reasonably efficient. The major task of this thesis is to develop coordination methods which fulfill the requirements of simplicity, generality, and efficiency for a simulation which is itself a packet communication architecture system.

## Outline of Thesis

In Chapter 2 methods of simulating the components of a packet communication architecture system, i.e. the modules and communication channels, will be discussed. First, rules of operation for packet communication architecture modules will be presented. Then, methods of simulating both the functional and timing aspects of the module will be developed. The emphasis will be on specifying what a correct simulation of a module would do, rather than on the more difficult problem of translating these requirements into actual programs. The problem of producing programs which will accurately simulate a module, based on some specification of the module, is left as an area for further research.

In Chapter 3 the ideas developed in Chapter 2 will be extended to allow the simulation of entire systems. As will be seen, if the simulation processors are simply loaded with programs which simulate the activities of the system components, the simulation might not accurately model the system but instead reach a deadlocked state. Besides simulating the activities of the modules, the simulation processors must communicate with each other to keep their efforts coordinated. The main purpose of this chapter is to develop methods of incorporating the coordination activities into the simulation processor programs. In this chapter a proof will be described which shows that the simulation will accurately model the actual system. The full proof is contained in Appendix 1. This proof demonstrates the benefits of the modular approach to the design of the simulation. First, the important requirements for the modules in the system and for the simulation programs of these modules will be specified. Second, it will be proved that the simulation and coordination methods of Chapters 2 and 3 satisfy these requirements. Finally, it will be proved that any simulation which satisfies the requirements will accurately model the actual system.

In Chapter 4 methods of terminating the coordination activities, once the modules in the system have ceased activity will presented. Without this termination, the simulation might run indefinitely, even though no module activities are being simulated. The last part of the chapter describes a proof of the correctness of the termination operations. The full proof is contained in Appendix 2. First, it is proved that these operations will not terminate the simulation too soon or in any other way interfere with the simulation operations. Hence, the requirements for the correctness of simulation proof will still apply. Then, it will be proved that the simulation will eventually terminate, if the actual system would terminate under the same circumstances.

In Chapter 5, the coordination methods of Chapter 3 will be further refined to increase the efficiency of the simulation. The coordination methods of Chapter 3 are designed to be very simple and uniform over all modules. As a result, the amount of coordination information passed between processors is high, and the concurrency of the processors' activities can be unnecessarily restricted. In some cases, the processor program for a module can be modified slightly to take advantage of specific properties of the module. Two examples of such modifications are presented. These two modifications will not increase the complexity or modularity of the simulation programs significantly but can greatly increase the efficiency of the simulation. Moreover, these modifications will not cause the simulation programs to violate any of the requirements for the correctness proof of Appendix 1 to apply. This further demonstrates the benefits of a modular approach to correctness proofs.

Finally, Chapter 6 contains conclusions, suggestions for other applications, and suggestions for further research. Some of the other applications include simulation of other types of systems, and application of the coordination and termination methods to other forms of distributed computation.

By working within the concepts of packet communication architecture, this thesis develops simulation techniques which fulfill the four design goals:

simplicity of hardware, generality, simplicity of software, and reasonable efficiency. Moreover, these techniques are provably correct. This is particularly comforting considering the subtle nature of parallel, asynchronous computations, which can often have unexpected deadlocks, races, nontermination problems, or other malfunctions.

For any computation which is designed to be executed by a parallel, asynchronous system such as a packet communication architecture system, a proof of correctness is essential. The traditional approach of implementing an initial version of a system and then debugging it will not work for computations which must be time-independent. Even if the computation is tested on a large number of test cases, one cannot be certain that it will be correct for all cases. A slight change in the timing of one part of the computation might lead to a deadlock, critical race, or other malfunction. Even in trying to prove the correctness, one can easily overlook some of the subtleties of the computation. However, by carefully developing a formal mathematical description of the computation and then proving that a computation which fulfills this description will operate correctly, these subtleties can be uncovered.

# Chapter 2

# Simulating the Components of a Packet Communication Architecture System

## Introduction

Each processor in the simulation must simulate the operations of one or more of the modules or communication channels in the actual system. This includes simulating the timing details of the module as well as the module's data operations. If the simulation is to itself be a packet communication architecture system, there can be no timing constraints on the simulation processors or on the communication links between processors. Hence, a method of simulating the timing must be developed which is independent of the speed of simulation.

## Module Operation

Before methods of simulating modules can be developed, the behavior which will be expected of these modules must be presented. In the interest of generality, these rules will be as unrestrictive as possible. As a result, some forms of behavior are allowed which are not quite in keeping with the philosophies of packet communication architecture design. However, as mentioned before, the designer of a system should not be restricted in the types of systems he can simulate. Furthermore, these allowances do not cause any added difficulties for the simulation.

At any time, a module is in one of two modes: the wait mode or the firing mode. While in the wait mode, the module cannot produce any output packets. Once the necessary conditions for firing are met, the module fires, meaning that it absorbs some of the input packets from its input ports, performs computations, and some time later sends packets from its output ports. Then it changes its internal state and reenters the wait mode. In general, an input port can be a buffer which can hold a number of packets simultaneously. A packet remains at an input port until it is absorbed by the module. An output port, on the other hand, is more like a door through which output packets pass.

The module must make the following decisions: when to fire, which input packets to absorb, what computations to perform, the values of the output packets and the times at which they are sent, and the new state of the module. These decisions can depend on the following factors:

1.) The values of all packets at the input ports.

2.) The time at which each of the input packets arrived.

3.) The current time.

4.) The current state of the module.

5.) Stochastic processes within the module.

However, while a module is in the firing mode, it cannot be affected by input packets which have arrived since the module entered the firing mode.

These rules of operation allow for modules whose behavior depends heavily on time: the current time of the module, and the time at which each input packet arrives. While this does not fit in well with the philosophy of time-independent design, it will not cause any particular difficulties for the simulation.

A packet communication architecture module has only three forms of input information:

1.) The initial state $S_0$ of the module.

2.) The values of the packets received at each input port.

3.) The time at which each input packet arrived.

Similarly, it produces only three types of output information:

1.) The final state $S_f$ of the module.

2.) The values of the output packets sent from each output port.

3.) The time at which each output packet is sent.

The output information produced by a module can depend only on the input information and the stochastic processes within the module. If the module contains no stochastic processes, then the simulation of the module should produce the correct output information based on the input information. If the module contains stochastic processes, then the simulation should produce the correct output information based on the input information and one set of choices for the random variables. Furthermore, the stochastic processes should be simulated in such a way that the values of the random variables are chosen with the same probability in the simulation as they would be in the actual module.

## Module History

The input and output information received and sent by a module while it is operating can be formally described in terms of histories. The history of a single port is a sequence of ordered pairs:

$$h = (x_1,t_1),(x_2,t_2),\ldots,(x_j,t_j),\ldots ,$$

where $x_j$ is the data value contained in the $j$th data packet arriving at or being sent from the port, and $t_j$ is the time at which it is received or sent. Since packets are sent or received one at a time, we have $t_j > t_{j-1}$, for all $j \geq 1$. We also require $t_1 > 0$. This implies that no output port can produce a packet at time 0. This restriction is part of the finite delay restriction which will be discussed in Chapter 3. Furthermore, no input port can receive a packet at time 0. Any packets present at an input port initially are considered part of the module's initial state, and not part of the input port's history.

While similar in idea, this definition of history differs from the definitions used by Patil [18] and Kahn [12] in their work with determinate systems. Their histories are sequences of data values only and contain no time values. Histories without time values were useful for them, since determinate systems have time-independent behavior. For simulation purposes, however, the simulation of the timing is as important as the simulation of the data operations. Moreover, the time values are part of the input and output information of the module. Hence, the time values are an important part of the history.

Since an infinite number of data packets could eventually pass through a port, a history can be an infinite sequence. However, for any physical system, there must be some minimum separation time $\delta$ between any two packets. Hence, no more than $t/\delta$ packets can pass through the port before time $t$. This implies that a history must be a countable sequence.

The history of an input port $i_k$ is denoted $hi_k$, and the history of an output port $o_k$ is denoted $ho_k$. The input history of a module $M$ with input ports $i_1,i_2,\ldots,i_n$ is the $n$-tuple of the histories of the input ports:

$$HI = \langle hi_1, hi_2, \ . \ . \ . \ , hi_n \rangle.$$

Similarly the output history of a module $M$ with output ports $o_1,o_2,\ldots,o_m$ is an $m$-tuple:

$$HO = \langle ho_1, ho_2, \ . \ . \ . \ , ho_m \rangle.$$

Just as the histories of the input ports to a module can be combined together, the histories of the system input ports (those input ports which receive packets from an external source rather than from other modules in the system) can be combined into a *system input history*

$$I = \langle hi_a, hi_b, \ . \ . \ . \ , hi_z \rangle,$$

where $i_a,i_b,\ldots,i_z$ are the system input ports. Similarly, the histories of the system output ports can be combined into a *system output history*

$$O = \langle ho_a, ho_b, \ . \ . \ . \ , ho_z \rangle,$$

where $o_a,o_b,\ldots,o_z$ are the system output ports.

It will be useful to define the relation "*is an initial segment of*" between two histories. First, a history $h_1$ is a *proper* initial segment of a history $h_2$, denoted $h_1 \sqsubset h_2$, if

$$h_1 = (x_1,t_1), (x_2,t_2), \ldots, (x_j t_j),$$

and either

$$h_2 = (x_1,t_1), (x_2,t_2), \ldots, (x_j,t_j), (t_{j+1},t_{j+1}), \ldots, (x_m,t_m),$$

or

$$h_2 = (x_1,t_1), (x_2,t_2), \ldots, (x_j,t_j), (t_{j+1},t_{j+1}), \ldots .$$

Then $h_1$ is an initial segment of $h_2$, denoted $h_1 \sqsubseteq h_2$, if $h_1 \sqsubset h_2$ or $h_1 = h_2$.

These relations can be extended to module input and module output histories as follows:

If

$$HI = \langle hi_1, hi_2, \ldots, hi_n \rangle$$

$$HI' = \langle hi'_1, hi'_2, \ldots, hi'_n \rangle$$

then $HI \sqsubseteq HI'$ if and only if:

$$hi_j \sqsubseteq hi'_j, \text{ for all } 1 \le j \le n.$$

The definitions for module output, system input, and system output histories are similar. Similarly, we can define the relation $\sqsubset$ over module and system histories.

A final notation is to define the history up to some time $t$. For a single port, $h(t)$ is a history, $h'$, where $h'$ contains all elements in $h$ with time values $\le t$. Hence $h(t) \sqsubseteq h$. This idea can be extended to module histories, as well:

$$HI(t) = \langle hi_1(t), hi_2(t), \ldots, hi_n(t) \rangle .$$

Thus $HI(t) \sqsubseteq HI = HI(\infty)$.

Using the notion of histories, the operation of a packet communication architecture module can be stated precisely. If the module contains no stochastic processes, then the output history HO and the final state $S_f$ are *functions* of the input history HI and the initial state $S_0$. For modules containing stochastic processes, HO and $S_f$ are functions of HI, $S_0$, and the values of the random variables.

Note that a module which computes a function over histories as they are defined here may not compute a function over the histories defined by Patil [18] and Kahn [12]. Since our histories include time values, modules such as arbiters and time clocks compute functions over these histories, whereas they are not functional over histories without time values.

## Channel Operation

In a packet communication architecture system, a communication channel serves only to carry the output packets from an output port of one module to an input port of another module. Furthermore, the channel preserves the ordering of packets. Packets will be received at the input port in the same order in which they sent from the output port. A channel's operations can be stated formally in terms of histories. If output port $o_p$ of module $M_1$ is connected to input port $i_r$ of module $M_2$, and $o_p$ has output history

$$\text{ho}_p = (x_1,t_1),(x_2,t_2),\ldots,(x_j,t_j),\ldots ,$$

then $i_r$ will have an input history

$$\text{hi}_r = (x_1,t'_1),(x_2,t'_2),\ldots,(x_j,t'_j),\ldots .$$

Due to the order preservation, $t'_j > t'_{j-1}$. Furthermore, since values cannot be received "before" they are sent, $t'_j \geq t_j$.

While a communication channel cannot change the values of data packets or their ordering, it can introduce a delay between the time at which they are sent and the time at which they are received. This delay must be simulated, since it will affect the input history of the module to which it is connected. The communication channel can be simulated by one of several means. First, we can simply ignore the delay and consider $\mathrm{hi}_r = \mathrm{ho}_p$. This would be appropriate in cases where the delay time of the channel is much smaller than the delay time of the modules. For example, if the modules are close together and directly wired to one another, the channel delay time will be very small. Second, we can simulate a module and the channels connected to its output ports as a single unit. Conceptually we can view this as extending the boundaries of a module $M$ to include its output channels, as shown in Figure 2.1. The output ports of this extended module $M'$ are wired directly to the input ports of other modules. This solution is appropriate if the channels connected to a module operate independently of other channels in the system, such as channels which are implemented as FIFO buffer units. Finally, the most general approach would be to simulate the channels as if they were packet communication architecture modules. This approach would be required if the channels do not operate independently of one another. For example, if packets are sent from one module to another over a network, such as the ARPA network [15], the delay time could depend on the total number of packets being sent over the network. In this case we would simulate the ARPA network as a

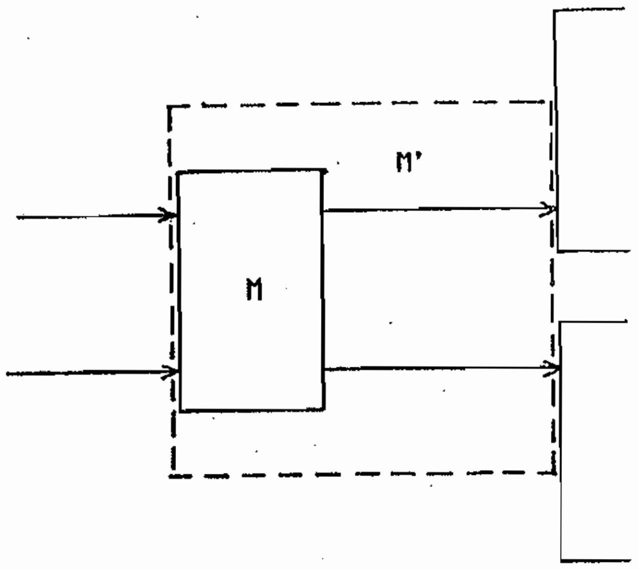


*Figure 2.1* - **Extending Module Boundary to Include its Output Channels.**

**********************************************************************

**packet communication architecture module.**

**For the remainder of this thesis, it will be assumed that the system to be simulated consists of a number of modules which are interconnected by zero-delay channels. Some of these "modules", however, might actually be extended modules or communication channels which are to be simulated. Thus, if output port $o_p$ of one module is connected to input port $i_r$ of another module, then $\mathbf{ho}_p = \mathbf{hi}_r$.**

## Time Independent Simulation of a Module

**The idea of a history leads quite naturally to a means of representing time in the simulation. The time at which a packet is sent from an output port can be considered part of its *value*, rather than an implicit property. Thus, the value of a packet is a pair $(x,t)$, where $x$ is its data value, and $t$ is its time value. By explicitly providing this time information in each packet, a**

simulation processor can simulate the operation of a module without any real-time constraints.

For example, suppose we wish to simulate a DIVIDE module as shown in Figure 2.2. If the simulation processor receives the packets, (x,10) and (y,20), on its input ports, then it will simulate the firing of the module at time 20, and, since the delay time of the module is 5, produce output packets $(x_{(\mathrm{mod}\ y)},25)$ and $(x/y,25)$. The simulation is not required to operate at a particular speed, since the actual time at which the output packets are sent during the simulation is not important.

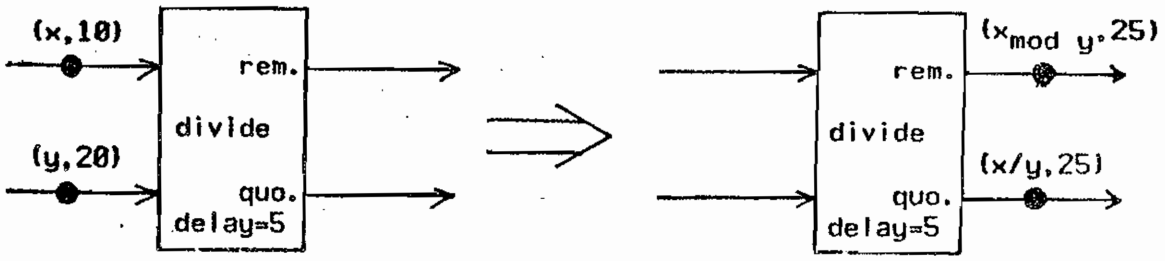


***Figure 2.2*** **- Example of Simulation Module Operation.**

With this means of simulating the timing, the *output* of the simulation of a module is the entire *output history* of the actual module. This can be described formally by defining *simulation histories*. For any port in the simulation, the simulation history is the sequence of packets passing through the port:

$$hs = (x_1,t_1),(x_2,t_2),\ldots,(x_j,t_j),\ldots ,$$

where $0 < t_1 < t_2 < \ldots < t_j < \ldots$ . If the simulation correctly simulates a port, then hs = h, where h is the history of the corresponding port in the actual system.

Simulation histories can be defined for modules, too. The *input simulation history* of a module is an $n$-tuple

$$HSI = \langle hsi_1, hsi_2, \ldots, hsi_n \rangle ,$$

and the output simulation history is an $m$-tuple

$$HSO = \langle hso_1, hso_2, \ldots, hso_m \rangle.$$

The system input simulation history SI and the system output simulation history SO are defined in a similar fashion. Furthermore, the relations $\sqsubseteq$ and $\sqsubset$ are defined over simulation histories in the same manner as they are over actual histories.

The requirements for the correct simulation of a module can be precisely defined in terms of histories for modules with non-stochastic behavior:

> Suppose an actual module produces an output history HO and finishes in a final state $S_f$ when it is started in some initial state $S_0$ and receives an input history HI. Then the simulation of this module must produce a simulation history HSO, such that HSO = HO, and it must finish in $S_f$, when it is started in state $S_0$, presented with a simulation history HSI = HI and then notified that no more input packets will be received.

The requirement that the simulation be notified when the last packet has been received is needed to prevent the simulation from *hanging up*, waiting for packets which will never arrive. This will be discussed later in this chapter.

Without any constraints on the times at which input packets arrive at the input ports of the modules in the simulation, there is no guarantee that the relative orderings of packets on different input ports will be preserved. This can lead to a problem of *premature firing*, in which the firing of a module at some time $tfire$ is simulated before all input packets with time $\leq tfire$ have arrived. For example, if an arbiter in the simulation receives a packet $(x,10)$ on one input port, it might simulate the firing at time $tfire = 10$, and (assuming it has a delay time of 2) send the packet $(x,12)$ from its output port. Suppose now, though, that a packet $(y,5)$ is received on its other input port. The arbiter has fired prematurely and the simulation cannot proceed properly.

To prevent this problem of premature firing, the firing of a module at time $tfire$ must not be simulated until the entire input simulation history $\mathrm{HSI}(tfire)$ has been received. The only way the simulation can know it has received $\mathrm{hsi}_k(tfire)$ on input port $i_k$ is if it receives a packet with time value $\geq tfire$ on that input port. Thus if the simulation stores the time value of the most recently received packet on each input port $i_k$, denoted $tlast_k$, then the firing of a module at time $tfire$ can be simulated if $tfire \leq \min_{1 \leq k \leq n} (tlast_k)$.

The simulation of a module proceeds as follows:

1.) Determine whether the module can fire at some time $tfire \leq \min_{1 \leq k \leq n} (tlast_k)$ based on the data and time values of those packets at the input ports with time values $\leq tfire$, the current state of the module $S_n$, and the outcome of simulations of any stochastic processes.

2.) If the module can fire, then simulate the firing of the module as follows:

a). Remove the proper input packets from each input port.

Only packets with time value $\leq$ *tfire* can be removed.

b). Calculate the output data values and the output times. These calculations can depend only on input packets with time values $\leq$ *tfire*. Furthermore, all output times must be greater than *tfire*.

c). Send the output packets from the proper output ports.

d). Calculate the new state $S_{N+1}$.

3.) Go to step 1.

Assuming the simulation will produce the proper output packets each time it simulates the firing of a module, the output of the simulation will always be an initial segment of the output history of the actual module, that is $HSO \sqsubseteq HO$. However, due to the requirement that $tfire \leq \min_{1 \leq k \leq n} (tlast_k)$, it is possible for the simulation of a module to *hang up* by waiting for packets which will never arrive. Suppose, for example, that an arbiter in the simulation receives a packet $(x,10)$ on input port 1 but has received no packets with time greater than 5 on input port 2. Then $tlast_k = 5 < tfire = 10$, hence the firing of the module cannot be simulated. If no more packets are ever received on input port 2, the firing of the module at time 10 will never be simulated, even though the module is enabled. The simulation must be notified somehow, when the last packet has been sent to each input port, so that any remaining input packets can be processed correctly. With this notification, the output of the simulation will be the output history of the actual module, in other words $HSO = HO$.

## Conclusion

By including the simulation time in each data packet, the operation of a module can be properly simulated without any real-time constraints. Although this requires each simulation processor to compute time values as well as data values, it enables us to simulate a wide variety of packet communication architecture systems with complete accuracy.

# Chapter 3

# Simulation of a System

## Introduction

In the previous chapter, methods of simulating the components of a packet communication architecture system were discussed. If, in an attempt to simulate the entire system, these module simulations were connected together, the simulation would most likely deadlock. This deadlock results when the modules in the simulation are waiting for packets from each other, but none can be fired until one of them produces more output packets. Unlike deadlocks which might occur in the actual system, which should be simulated, this form of deadlock, called *hanging up*, prevents the simulation from fully simulating the activities of the actual system.

For example, the simulation program for the arbiter in Figure 3.1 has received a packet with time 3 on input port 2, but nothing on input port 1. Hence $tlast_1 = 0 < tfire = 3$, and the firing of the arbiter cannot be simulated. However, no packet will ever be received on the other input port until the adder module fires, but this will not happen until the arbiter fires. The simulation has hung up. The actual system would not have deadlocked under these circumstances, though. The arbiter would have fired and sent the packet ($y$) at time 5 to the adder, which would have fired at time 10, and so on. The simulation has ceased operation at an earlier time than the actual system would have. A proper simulation would reach the same state that the actual system would. Additional coordination between the processors is needed to prevent the simulation from hanging up.

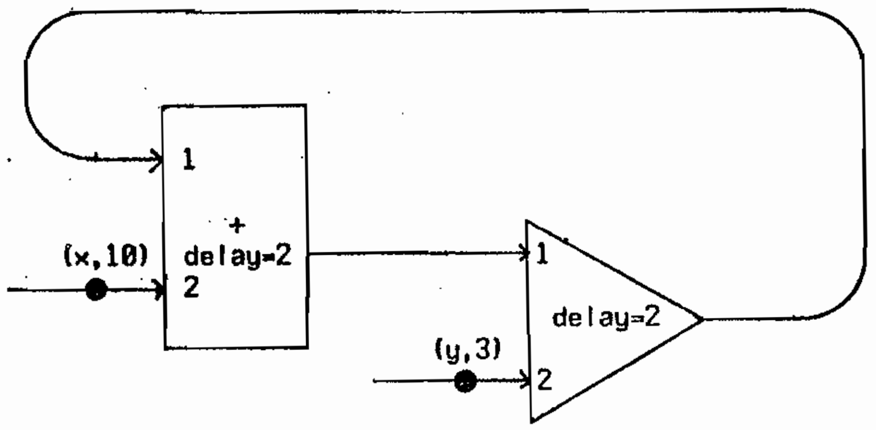


*Figure 3.1* - Simulation which has "Hung Up."

In this chapter, a means of providing this coordination will be presented which preserves the principles of packet communication architecture, including: autonomy of modules, communication by packets, and time-independence. One further feature of this coordination method is that all coordination information is sent along the same paths as the data packets are. There is no need for additional communication links between processors.

For each module to be simulated, a simulation processor must perform two types of operations: module activity simulation, and coordination. These operations together comprise the activities of a process called the *simulation module*. If the simulation is itself to be a packet communication architecture system, each simulation module must be a packet communication architecture module. This means that the simulation modules can be viewed as autonomous processes, even if several of these processess are executed by one simulation processor.

## Coordination Algorithm

The simulation hangs up when the simulation modules fail to communicate their status to each other but instead wait passively for other simulation modules to take action. Instead, the simulation modules could send status information to each other in the form of *time packets* of the form $(t)$, where $t$ is a time value. Time packets are sent along the normal communication links between simulation modules. When a simulation module sends a time packet $(t)$ from an output port, this indicates that no packets with time values less than or equal to $t$ will be sent from this output port in the future.

At any point in the simulation that a module is in the wait mode, if there is no value of $tfire \le tmin = \min_{1 \le k \le n} (tlast_k)$ for which the module can fire, then the module cannot possibly fire before or at time *tmin*. If the module has a minimum delay time *delay* between firing and producing the first output packets, then the ***minimum output time*** is given by the formula:

$$\begin{aligned} tout &= tmin + delay \\ &= \min_{1 \le k \le n} (tlast_k) + delay. \end{aligned}$$

The simulation module cannot produce more output data packets with time values less than or equal to *tout*, hence time packets $(tout)$ can be sent from all output ports which have not already produced packets with time values greater than or equal to *tout*. Furthermore, if the firing of a module at some time *tfire* is simulated, but no data packets are sent from an output port $o_j$ then a time packet $(tfire+delay)$ can be sent from $o_j$, since any future data packets from this output port will have time values greater than *tfire* + *delay*.

As long as all time and data packets are sent from each output port of a simulation module with strictly increasing time values, and the communication links between the simulation modules preserve the ordering of the packets, the value of $tlast_k$ for an input port is still the last time value received on that input port, either as part of a data packet or as a time packet. No new packets can be received at an input port with time values less than or equal to $tlast_k$. If the values of *delay* are greater than zero for all simulation modules, then as a result of these coordination activities, the simulation modules will send increasingly larger time values to one another, until one of the simulation modules is able to simulate the firing of its module, thereby avoiding deadlocks.

In the example shown in Figure 3.1, The simulation module for the arbiter has received a data packet with time value 3 on input port 2 and has received nothing on input port 1. The arbiter cannot possibly fire before time $tmin = \min(tlast_1, tlast_2) = \min(0,3) = 0$. Hence it cannot produce any output packets with time value less than or equal to $tmin + delay = 0+2 = 2$. Therefore it can send a time packet (2) to input port 1 of the adder's simulation module which in turn would update $tlast_1$ to 2. The adder cannot possibly fire before time $tmin = \min(2,10)$ and therefore cannot produce any output data packets with time values less than or equal to $tmin + delay = 2+2 = 4$. Therefore a time packet (4) can be sent back to the arbiter's simulation module which would then set $tlast_1 = 4$, and, since $tfire = 3 \leq \min(tlast_1, tlast_2) = \min(4,3)$, the firing of the arbiter module would be simulated.

The operation of a simulation module can be stated as follows:

1.) Each time a time or data packet is received on input port $i_k$, update $tlast_k$.

2.) Determine whether the module can be *safely* fired. That is, whether the conditions are sufficient for the module to fire at some time *tfire*, where

$$tfire \leq \min_{1 \leq k \leq n} (tlast_k).$$

a.) If the module can be safely fired, then simulate the operation of the module on those input packets with time values ≤ *tfire* and produce the output data packets. For each output port $o_j$ from which data packets are sent, update the value of $tlast\text{-}out_j$, which is the time value of the most recently sent output packet from $o_j$. For each output port $o_j$ for which $tlast\text{-}out_j < tfire + delay$, send a time packet $(tfire + delay)$ from $o_j$ and update $tlast\text{-}out_j$.

b.) If the module cannot be safely fired then compute *tout* where

$$tout = tmin + delay,$$

and send a time packet (*tout*) from each output port $o_j$ for which $tout > tlast\text{-}out_j$. Then update the value of $tlast\text{-}out_j$ for each of these output ports. The value of *delay* must be greater than zero but cannot be greater than the minimum time required for the module to produce an output packet after firing.

3.) Return to step 1.

These coordination operations are quite simple, especially since time packets are produced primarily when the simulation module is otherwise inactive. The simulation module must store the value of $tlast_k$ for each input port, and $tlast\text{-}out_k$ for each output port. However, no storage for time packets is required since they are not needed once the values of $tlast_k$ have been updated.

Furthermore, the simulation requires some means of determining when the system input ports have received their final data packets. For instance, in the example shown in Figure 3.1, the firing of the arbiter at time 3 would be simulated and the packet (y,5) would be sent to the adder's simulation module, as shown in Figure 3.2.

********************************************************************

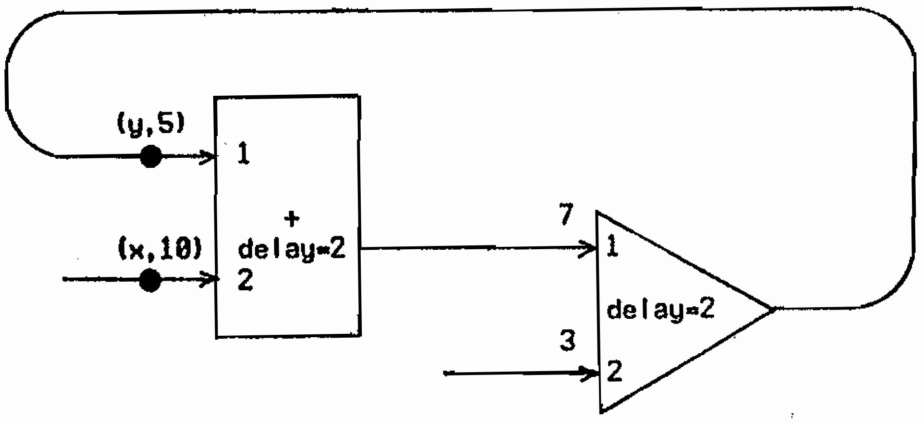


***Figure 3.2* - Simulation Requiring Packets on System Input Ports.**
**The numbers alongside the input ports represent the values of *tlast* for the ports.**

********************************************************************

Suppose that no more packets are received at input port 2 of the arbiter (this is a system input port.) Then the adder module will be enabled to fire at time *tfire* = max(5,10) = 10, but the simulation module cannot simulate this firing, since $tlast_1$ = 5 < *tfire* = 10. Instead, a time packet with value min(5,10) + 2 will be sent to the arbiter's simulation module. This simulation module will compute *tout* = min(7,3) + 2 = 5, hence no time packet will be sent. Once again, the simulation has hung up. The simulation module for the arbiter is still expecting data packets on input port 2, but none will ever arrive. In order for a simulation to complete all operations up to some time *tfinal* time packets with value ≥ *tfinal* must be sent to all system input ports after the last data packets have been sent. If we want to simulate the entire operation of the system, time packets with value ∞ must be sent to all system input ports, where ∞ i greater than any other time value. This can lead to a nonterminatin simulation in which the simulation modules keep sending time packets to on another indefinitely, even though no modules will ever be enabled to fire again A means of terminating the simulation will be presented in Chapter 4.

In our example, we want to complete all operations with time $\leq$ 10. If time packet (10) is sent to the arbiter's simulation module, it would comput $tout$ = min(7,10) + 2 = 9 and send this value to the arbiter. The adder stil cannot be fired safely, but a time packet with value min(9,10) + 2 = 11 woul be sent back to the arbiter's simulation module which in turn would send bac a time packet with value min(11,10) + 2 = 12. Finally, $tfire$ = 10 min($tlast_1$,$tlast_2$) = min(12,10), and the firing of the adder at time 10 could b simulated.

With the addition of time packets, the simulation histories contain mor than just data packets. When comparing simulation histories to actual historie however, only the data packets are of interest. The function data is applied t simulation histories to give the sequence of data packets (including their tim values) contained in a simulation history. For example, if

$$\text{hs} = (x,1),(3),(y,30),(z,35),(100),$$

then

$$\text{data(hs)} = (x,1),(y,30),(z,35).$$

The function data can be applied to module simulation histories and syste simulation histories as well.

## Features of the Coordination Algorithm

This coordination algorithm preserves the philosophies of packet communication architecture design. All coordination information is passed between simulation modules in the form of time packets. There are no time constraints on the simulation modules, and the simulation modules can operate independently. Furthermore, the coordination operations for each module are very simple. Each simulation module performs identical coordination operations, which allows uniformity in the simulation programs.

One further feature is that a simulation module sends time packets only to those simulation modules to which it also sends data packets, and these time packets are sent over the normal data paths. This not only keeps the number of input and output ports to a simulation module limited, it eliminates the need to synchronize the coordination information with the data information. If, on the other hand, time packets were sent along some other communication links, special measures would be required to prevent a time packet from arriving at an input port before a data packet having an earlier time value does. By sending time packets along the normal communication links, we use the first-in, first-out property of these links to ensure the proper sequencing of time and data packets.

## Efficiency of Coordination

This coordination algorithm is rather inefficient in two respects. First, a large number of time packets must be sent to keep the simulation coordinated. In the example of Figures 3.1 and 3.2, a total of seven time packets were transmitted so that the arbiter and the adder could each fire once. This causes both a delay in the simulation and a heavy load on the communication channels between simulation modules. For larger simulations, the number of time packets would be overwhelming. Second, this method does not allow all possible concurrency in the simulation. For example, the two modules shown in Figure 3.3 could potentially be simulated at the same time. The adder will not fire until time 10 and hence cannot produce a packet with time < 12. Therefore, the firing of the arbiter at time 11 could be simulated at the same time as the firing of the adder. With the coordination algorithm described, however, the simulation module for the arbiter would receive a time packet with value min(5,10) + 2 = 7 and hence the arbiter would not be simulated until after the adder has been simulated. This lack of concurrency compromises the efficiency of the simulation, since it causes the simulation processors to wait unnecessarily.

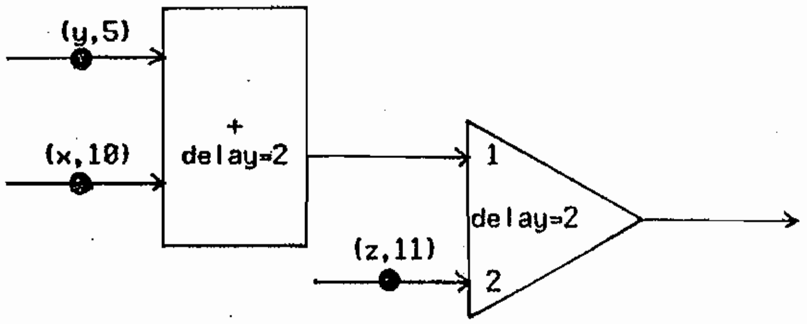


*Figure 3.3* - Modules which can be Simulated Concurrently.

This inefficiency could be reduced if more use were made of the specific properties of the modules being simulated. With the coordination algorithm described only two properties are assumed about the modules to be simulated: they will not produce any output packets while in the wait mode, and for each module there is some minimum delay time *delay* between when it fires and when it produces the first output packets. This, of course, makes the coordination procedures very simple, but it creates the two inefficiencies mentioned above. If, on the other hand, we make use of the fact that an ADD module cannot fire without first receiving data packets on *both* input ports, then for the example in Figure 3.1, the earliest possible time for it to produce an output packet could be calculated as

$$\begin{aligned} tout &= \underline{\max}(tlast_1, tlast_2) + 2 \\ &= \max(0,10) + 2 = 12. \end{aligned}$$

The time packet (12) could be sent to the arbiter's simulation module which would then fire the arbiter at time 3 and send the packet (y,5) to the adder's simulation module. Furthermore, an ADD module can only absorb one data packet at a time from each input port, hence the firing of the module at time 10 could be simulated even though $tlast_1 = 5 < tfire = 10$. By making use of these two particular properties of ADD modules, only one time packet would be transmitted in the simulation, as opposed to the original seven.

Of course, there is a trade-off between the complexity of the coordination procedures within each simulation module, and the efficiency of the coordination. In the most extreme case, each simulation module could simulate the entire system internally to determine whether a particular module can be safely fired. This would certainly minimize the amount of coordination information sent between simulation modules, but it would be overwhelmingly complex. In Chapter 5, several refinements to the proposed coordination method will be described. The emphasis will be on refinements which do not increase the complexity much but do increase the efficiency significantly.

## Correctness of the System Simulation

The combination of the module activity simulation and the coordinatio operations for each module will guarantee that when the simulation modules ar interconnected, they will accurately model the activities of the actual system A proof of this is presented in Appendix 1 and will be described briefly here.

The proof applies only to modules whose output history and final state ar functions of the input history and initial state. The module cannot contain an stochastic processes. However, for a particular set of choices of random variables, the output history and final state of a module will always b functions of its initial state and input history, in which case the proof wil apply. If the stochastic processes are simulated in such a way that the rando variables are chosen with the same probability as they would be in the actua system, the simulation will stochastically model the actual system.

To formally describe the operations of the actual modules and th simulations of these modules, six requirements are specified: three for the actua modules and three for the simulations of these modules.

For the actual modules, the requirements are:

1.) **Functionality of Output:** The output history and final state of module depends only on the initial state of the module and the input history.

2.) Monotonicity of Output: The output of a module at time *t* cannot be affected by input received after time *t*.

3.) Finite Delay: The output of a module at time *t* cannot be affected by input received *at* time *t*. In other words, there must be a finite delay between the receipt of an input packet and the production of an output packet which depends on this input packet.

If a module satisfies all three of these requirements, then the output history of the module up to and including time *t* is a function of the initial state and the input history up to but not including time *t*.

These three requirements for the modules to be simulated are not very restrictive. The monotonicity of output requirement simply implies that a module cannot look into the future and predict what input will arrive, nor can it retract or alter any output packets once they have been sent out. The finite delay requirement states that a module cannot react instantaneously to an input packet. This is true for any physically realizable module. The functionality of output requirement implies that the module cannot receive any input information other than the initial state and packets arriving at the input ports. Furthermore, the module cannot contain any stochastic processes, unless we consider the operation of the module for a particular choice of random variables.

For the simulation of each module the requirements are:

1. Correct Module Simulation: The simulation of a module must produce the same data packets with the same time values as the actual module would for the same input conditions. That is, suppose the simulation of a module produces a simulation history HSO when it starts in initial state $S_0$ and receives an input simulation history HSI where all of the data and time packets arriving at each input port have strictly increasing time

values. Let

$$tfinal = \min_{1 \le k \le n} (tlast_k)$$

after the input simulation history HSI has been received. That is, *tfina* is the smallest of all the final time values received by the input ports o the simulation module. Then

$$data(HSO(tfinal)) = HO(tfinal),$$

where HO is the output history of the actual module when it starts i the same initial state $S_0$ and receives the input history HI = data(HSI) Furthermore, if *tfinal* = ∞ (all input ports to the module receive tim packets with value ∞), then the final state $S_f$ of the simulation of th module will be the same as the final state of the actual module.

2.) Correct Ordering of Output Packets: If the packets arriving at eac input port of a module in the simulation have strictly increasing tim values, then the output packets sent from each output port of the modul in the simulation will have strictly increasing time values.

3.) Correct Coordination: If a simulation module receives an inpu simulation history HSI then if *tfinal* = $\min_{1 \le k \le n}$ $(tlast_k)$, eventually a time o data packet with time value *greater* than *tfinal* will be sent from eac output port of the simulation module, unless *tfinal* = ∞, in which cas time packets with value ∞ will be sent from all output ports if th corresponding actual module ever terminates.

The first step in the correctness proof is to show that the simulation an coordination operations which have been developed will fulfill the thre requirements for the simulation modules. Then, it is proved that for *an* simulation in which the actual modules satisfy their three requirements and th simulation modules satisfy their three requirements, the simulation wil accurately model the actual system. This is stated in the following theorem:

---

__Theorem 1__. Correctness of Simulation.

Suppose a simulation has the following properties:

1.) The modules to be simulated satisfy the monoticity of output, finit delay, and functionality of output requirements.

2.) The simulation of each module satisfies the correct module simulation, correct ordering of output packets, and correct coordination requirements.

3.) All communication links between simulation modules operate properly, so that if input port $i_k$ is connected to output port $o_r$ then $\mathrm{hsi}_k = \mathrm{hso}_r$.

4.) The simulation receives a system input simulation history SI, and the sequence of time values received at each system input port is strictly increasing.

Let
$$tfinal = \min(tlast_a, tlast_b, \ldots, tlast_z),$$
after the system input simulation history SI has been received, where $i_a, i_b, \ldots, i_z$ are the system input ports. Then the simulation module for any module $M_j$ in the system will produce a module output simulation history $HSO_j$ such that

$$\mathrm{data}(HSO_j(tfinal)) = HO_j(tfinal),$$

where $HO_j$ would be the output history of the corresponding module in the actual system under the following conditions:

1.) All modules in the actual system are started in the same initial state as the corresponding simulation modules.

2.) The actual system receives the system input history I where
$$I = \mathrm{data}(SI).$$

Furthermore, if $tfinal = \infty$, the final state of each simulation module which terminates will equal the final state of the corresponding module in the actual system.

---

The theorem is proved by induction on the sequence of time values

$$t_0, t_1, t_2, \ldots, t_l, \ldots ,$$

where $t_0 = 0$, and

$$t_0 < t_1 < \ldots < t_l < \ldots \leq \infty,$$

and each time value $t_l$, $l > 0$, is contained in some actual or simulation history for the system. That is, $t_l$ is contained in one of the following histories: I, the system input history to the actual system, $HO_j$, the output history of some module $M_j$, SI, the system input simulation history, or $HSO_j$, the output simulation history of some module $M_j$.

The induction hypothesis is as follows: For any $t_l \in t_0, t_1, \ldots, t_l, \ldots$ such that $t_l \leq tfinal$,

a.) $data(HSO_j(t_l)) = HO_j(t_l)$, for all modules $M_j$, *and*

b.) Either $t_l = \infty$, or for any output port $o_r$:
$$hso_r(t_l) \sqsubset hso_r.$$

In other words, not only will the simulation accurately model the actual system up to and including time $t_l$, but in addition the coordination operations will cause each simulation module to send packets with time values greater than $t_l$ from all of its output ports. Thus the simulation cannot hang up due to a simulation module waiting for an input packet with time value $\leq t_l$, as long as $t_l \leq$ *tfinal*. Therefore, by induction, the simulation will accurately model the actual system up through time *tfinal*.

## Conclusion

By incorporating some relatively simple coordination operations in the simulation modules, the simulation will accurately model the actual system while preserving the properties of a packet communication architecture system. As a result, however, the simulation might fail to terminate even if the actual system terminates, and the simulation will be rather inefficient. These two difficulties will be dealt with in the next two chapters.

# Chapter 4

# Termination of the Simulation

## Introduction

Due to the decentralized and time-independent nature of the simulation and coordination operations, there are conditions for which the actual system will eventually cease all operation, but the simulation will continue indefinitely. The simulation modules can keep sending time packets with increasingly larger time values to each other long after all module activity simulations have been completed.

For example, in system of Figure 4.1 the system input port (input port 2 of the arbiter) has received a time packet with value ∞ and the simulation module for the switch has produced a data packet (x,97). As can clearly be seen, all data operations by modules in the system have been completed. The simulation, however, will keep going. The arbiter will send a time packet with value min(100,∞)+1 = 101 to the functional operator. This operator will send a time packet with value 101+2 = 103 to the switch, which will send a time packet with value 103+1 = 104 to the next operator. This operator, in turn, will send a time packet with value 104+3 = 107 to the arbiter. Then the arbiter's simulation module will start this cycle over again, even though nothing is really being simulated.

In this chapter, termination operations which can be incorporated in the simulation modules will be developed. These terminations operations guarantee

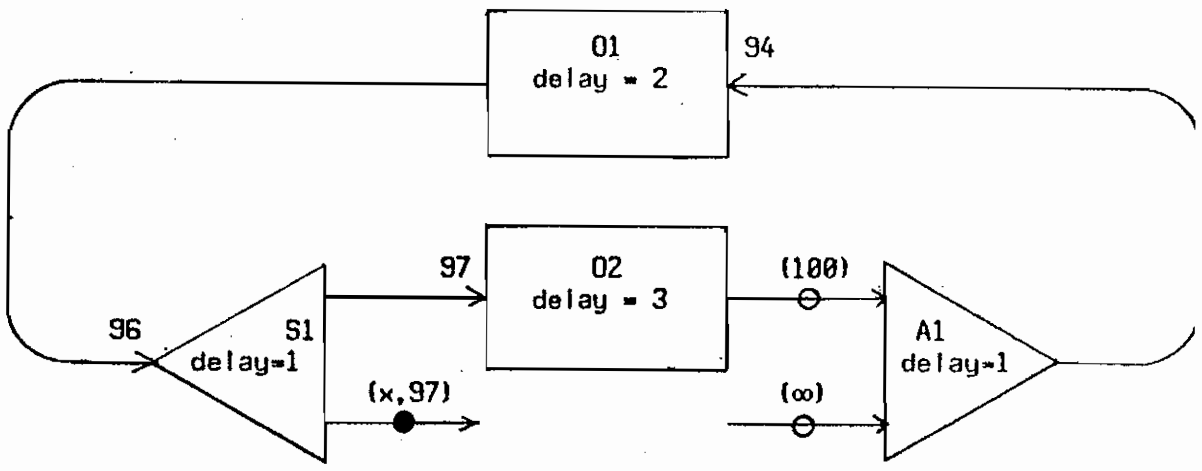


***Figure 4.1* - Nonterminating Simulation.**
**The circles represent time packets; the dots represent data packets; and the numbers alongside input ports represent the values of *tlast* for the input ports.**

**that the simulation will eventually terminate if the actual system does, while preserving both the correctness of the simulation and the principles of packet communication architecture. Furthermore, as with the coordination, all control information is sent between simulation modules along the normal data paths. No special hardware is required for termination, only additions to the simulation programs. The last part of this chapter describes a proof of correctness for the termination operations. The full proof is included in Appendix 2.**

**If there were some means of simultaneously observing all simulation modules and all communication links between them, then it could be determined when the simulation has completed all data operations. The simulation has completed all data operations and can be safely terminated once it reaches a point where: all system input ports have received time packets with value ∞, no modules have sufficient data packets to fire, and there are no data packets in**

transit between the simulation modules. This omniscient observer, however, would not be in keeping with the philosophies of packet communication architecture design. For our simulation, the simulation modules must send control information to each other to determine whether the termination conditions are satisfied. Furthermore, these termination operations must be time-independent.

Most of the standard methods of determining whether a system is active, such as time-outs, or waiting for a maximum count on the number of time packets will not work for this simulation. There are, however, special features of packet communication architecture modules which can be taken advantage of.

## Connectivity Classes

A module $M_2$ can only receive input information in the form of packets arriving at its input ports. Hence if there is no path from module $M_1$ to $M_2$, then the activities of $M_1$ cannot affect those of $M_2$. To make use of this idea, the meaning of *path* must be defined more formally. First, a module $M_1$ "*is connected to*" a module $M_2$ denoted $M_1 \rightarrow M_2$, if an output port of module $M_1$ is connected to an input port of $M_2$. There is a *path* from a module $M_1$ to a module $M_2$, denoted $M_1 \xrightarrow{+} M_2$, if there exists a sequence

$$M_1, M_a, M_b, \ldots, M_z, M_2,$$

such that

$$M_1 \rightarrow M_a \rightarrow M_b \rightarrow \ldots \rightarrow M_z \rightarrow M_2.$$

All communication with a module is in the form of data packets travelling along data channels. Hence if there is no path from $M_1$ to $M_2$, then there is no way for $M_1$ to send information to $M_2$, either directly or indirectly.

The difficulties in terminating the simulation arise when the systen contains cycles. A module is contained within a cycle if there is a path fror one of its output ports to one of its input ports, that is $M_j \xrightarrow{+} M_j$. Fo example the system of Figure 4.1 has a cycle O1 $\rightarrow$ S1 $\rightarrow$ O2 $\rightarrow$ A1 $\rightarrow$ O The simulation modules contained in cycles will not normally terminate - the cannot send time packets ($\infty$) until time packets ($\infty$) have been received on a input ports, but the simulation modules will not receive these time packe unless they send them out. Instead, the simulation modules will keep sendin time packets with values less than $\infty$ around the cycles indefinitely.

The cycles in the system can be identified by looking at the equivalen classes formed by the relation $\xleftrightarrow{+}$ where $M_1 \xleftrightarrow{+} M_2$ if and only if either $M_1$ $M_2$ (they are the same module), or $M_1 \xrightarrow{+} M_2$ and $M_2 \xrightarrow{+} M_1$. This relation indeed an equivalence relation [17]: it is reflexive, symmetric, and transitiv Hence it defines a set of equivalence classes which are called *connectivi* *classes* and are denoted $C_1, C_2, \ldots, C_q$. For any connectivity class containi more than one module, any two modules in the class must have paths to ea other. That is, if $M_1$, $M_2 \in C_j$, then

$$M_1 \xrightarrow{+} M_2 \text{ and } M_2 \xrightarrow{+} M_1.$$

An example of a system divided into its connectivity classes is shown in Figu 4.2.

The relation $\xrightarrow{+}$ can be extended to connectivity classes. $C_i \xrightarrow{+} C_j$ if a

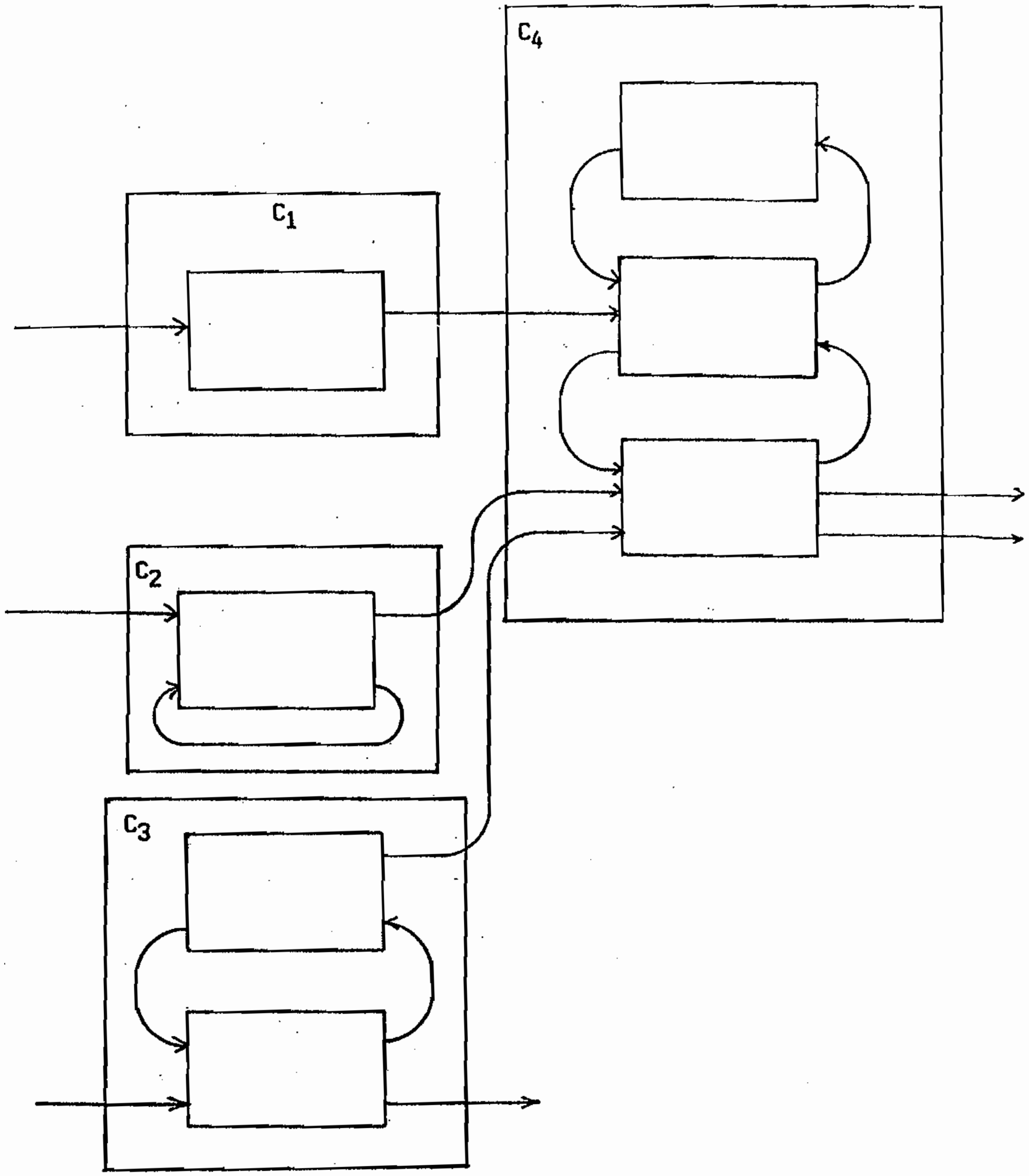


***Figure 4.2* - System Divided into Connectivity Classes.**

only if $M_i \xrightarrow{+} M_j$ for every $M_i \epsilon C_i$, $M_j \epsilon C_j$. In fact if $M_i \xrightarrow{+} M_j$ for *any* $M_i \epsilon C_i$, $M_j \epsilon C_j$, then $C_i \xrightarrow{+} C_j$. Moreover, if $C_i \xrightarrow{+} C_j$, then $C_j \not\xrightarrow{+} C_i$, or else they would not be separate equivalence classes. Thus, if $C_i \xrightarrow{+} C_j$, then the modules in $C_i$ are not affected in any way by the modules in $C_j$. We can terminate the modules in $C_i$ without worrying about the modules in $C_j$.

Using the properties of connectivity classes, the conditions for terminating a connectivity class $C_j$ can be stated. When all of these conditions are satisfied, the simulation modules in the class can safely terminate.

1a.) All system input ports which are input ports to modules in $C_j$ have received time packets with value $\infty$.

1b.) All classes $C_i$ such that $C_i \xrightarrow{+} C_j$ have been terminated.

2.) No module $M_j \epsilon C_j$ has sufficient data packets to fire.

3.) None of the channels connected to input ports of the simulation modules in $C_j$ contain data packets.

If there were some means of detecting when a connectivity class could be terminated, then all simulation modules in the class could send out time packets $(\infty)$ from all of their output ports. In this case, termination conditions 1a.) and 1b.) would be identical, from a connectivity class' point of view. That is, an input port $i_k$ to a module $M_j \epsilon C_j$ receives packets from one of three sources: a source external to the system, a module $M_i \epsilon C_i$ where $C_i \xrightarrow{+} C_j$, or a module $M_l \epsilon C_j$. In the first case, $i_k$ is a system input port and hence would receive a time packet with value $\infty$. In the second case, the input port $i_k$ would receive a time packet with value $\infty$ once the connectivity class $C_i$ has been terminated. Conditions 1a.) and 1b.) can therefore be restated as:

1.) Time packets with value $\infty$ have been received on all those input ports of modules in the class $C_j$ which are not connected to output ports of other modules in the class.

No special communication other than time packets is needed between connectivity classes or with the external world for termination. All that is needed to terminate the simulation of a system is some means of detecting when the modules in each class can be terminated.

If a class $C_j$ contains only a single module $M_j$ then this module either is not contained in any cycle in the system, i.e. $M_j \not\rightarrow M_j$, or it is part of a self-loop, in which there is a channel connecting an output port of the module to an input port of the module, so that $M_j \rightarrow M_j$. In the first case, the normal coordination operations of the simulation module are sufficient for termination. Since no input ports to the module are connected to output ports of modules in the class, time packets with value $\infty$ will eventually be received on all input ports of the module. The firing of the module at any time $\leq \infty$ will then be simulated. Then, since *tout* $= \infty$, time packets $(\infty)$ will be sent from all output ports, and the simulation processor can terminate the simulation of this module. Thus, no special termination procedures are required for modules which are not part of a cycle in the system.

For modules which are part of a self-loop and for connectivity classes with more than one module, however, the normal coordination operations are not sufficient for terminating the module simulations. For example, the modules in Figure 4.1 are all in the same connectivity class and therefore would not terminate. Those input ports which are connected to output ports of modules in the class will never receive time packets with value ∞ without special termination procedures.

## Termination Algorithm for Connectivity Classes Containing Cycles

A means of incorporating termination operations into the simulation module for each module in a connectivity class $C_j$ will now be given. This termination algorithm requires no changes in the topology of the system. There is no need to add more modules or communication links to the system. Unlike the coordination operations, the termination operations are not identical for each simulation module. First, one of the modules in the class is designated as the *termination control module*, denoted T, for the class. Any of the modules in the class can be chosen for this role. The simulation module for this module must initiate and validate the tests for completion of all operations by the modules in the class. Next, for each module in the class other than T, one of the output ports of the module must be selected as the signal output port of the module. These signal output ports must be selected in such a way that if we look only at the modules in the class, there is a path from every module to T following only channels connected to the signal output ports of the modules. Finally, for each module in the class, we must determine which input and output ports are connected to output and input ports of other modules in the class. The set of all input ports of $M_j$ which receive packets from modules in the class is denoted $\text{from\_class}_j$. Similarly, the set of output ports of $M_j$ which send packets to other modules in the class is denoted $\text{to\_class}_j$.

The termination operations for the simulation module of the termination control module T are as follows:

1.) Perform normal simulation and coordination activities until every input port which is not in $\text{from\_class}_T$ has received a time packet with value ∞.

2.) When there is no way for the module to fire without receiving more data packets, send *test* packets (test.+) from all output ports in $\text{to\_class}_T$.

3.) Wait until K test packets have been received on the input ports, where

$$K = 1 + \sum_{M_i \in C_j} (|\text{to\_class}_i| - 1).$$

In this equation, $|\text{to\_class}_i|$, is the number of output ports of module $M_i$ which are connected to input ports of other modules in the class.

4.) If any data or time packets are received while waiting for the test packets, continue with the simulation and coordination operations for the module.

5.) Determine the validity of the test as follows:

a.) If all K test packets have value test.+, and no data packets were received while waiting for the test packets, then send time packets (∞) from *all* output ports of the module.

b.) If at least one of the returning test packets has value test.- or a data packet was received while waiting for the test packets, then send packets (reset) from all output ports in $\text{to\_class}_T$, wait for K (reset) packets to return, and go to step 1.

6.) Once time packets (∞) have been received on all input ports of the simulation module, terminate the simulation of the module.

For every other module $M_j$ in the class, the termination operations for the simulation module are as follows:

1.) Perform normal simulation and coordination operations until a test packet is received on some input port.

2.) When the first test packet is received, continue simulating the module until all input ports which are not in $from\_class_j$ have received time packets with value ∞, and the data packets present at the input ports are not sufficient for the module to fire. Then, if the test packet has value test.+, and no data packets have been received since the test packet arrived, send (test.+) packets from all output ports in $to\_class_j$. Otherwise send (test.-) packets from all output ports in $to\_class_j$.

3.) If the module receives any more time or data packets, then continue the simulation and coordination operations as before.

4.) Any time another test packet arrives, if the packet has valu test.+, and no data packets have been received since the previou test packet was sent, then send a (test.+) packet on the signa output port. Otherwise send a (test.-) packet on the signal outpu port.

5.) When the first (reset) packet is received on an input port send a packet (reset) from each output port in $to\_class_j$ an prepare for a new test. If any further (reset) packets are receive before the next test, send them from the signal output port. Whe new test packets arrive, return to step 2.

6.) When a time packet (∞) is received on *any* input port i $from\_class_j$, send packets (∞) from all output ports, unless this h already been done.

7.) Once time packets with value ∞ have been received on a input ports to the module, terminate the simulation of the module.

During the course of a test, unless some simulation module can never t terminated, a test packet will travel through every communication link betwee the simulation modules in the class. Hence, every simulation module wi receive at least one test packet. Initially, T sends out $|to\_class_T|$ test packet On receipt of its first test packet, a simulation module $M_i$ will send o $|to\_class_i|$ test packets, thereby "creating" $|to\_class_i| - 1$ new test packet Thereafter, it will simply pass a test packet from an input port to an outp port. Hence, a total of K test packets will be created. The values of these test packets will be test.+ only if no form of data activity is found anywhere in the class. Because of the way in which the signal output ports are chosen, all K test packets will be funneled back to T which can then check the test results.

## Features of the Termination Operations

This termination algorithm preserves most of the desirable properties of the coordination algorithm. In particular, the simulation modules still fulfill the requirements for a packet communication architecture system. Although one module in each class is denoted as a termination control module, its only function is to initiate and collect information about each test. This module has no ability to monitor other modules or exercise any active control. Hence, the simulation modules are still autonomous. Furthermore, all communication is by packets, and the operations do not depend on any timing restrictions.

As with the coordination algorithm, all termination control information is sent over the normal data channels. This avoids the problem of monitoring the communication links between simulation modules. Instead, the first-in, first-out property of these links ensures that no data packets will be overlooked while they are travelling between simulation modules. No special hardware is required for termination operations, only additions to the simulation modules.

One undesirable feature of these termination operations is their dependence on the overall structure of the system to be simulated. Whereas the simulation and coordination operations of a module depend only on the module itself, the termination operations depend on how the module is incorporated in the system. This compromises the modularity of the design somewhat. However, the termination operations of a module can be fully determined based on a very limited amount of knowledge about the system, namely how modules in the system are interconnected. No details about the operations of other modules in the system are required. Thus, while the incorporation of the termination operations into the simulation modules will decrease the modularity of design, this decrease will be rather small.

## Efficiency of the Termination Operations

The termination opertations for the modules in a connectivity class are designed to be both simple and efficient. That is, they will not increase the complexity of the simulation modules greatly, nor will the speed of the simulation be decreased greatly. The efficiency is a result of several important features. First, the simulation and coordination operations need not be interrupted while the termination operations are taking place. Thus, if a test is initiated while modules in the class are still active, the simulation can keep going, although at a slightly decreased speed. Second, the operations are designed to keep the number of tests initiated reasonably low. The first test can be initiated as soon as the termination control module has received packets $(\infty)$ on all input ports which are not in from_class$_T$. However, all $K$ returning test packets will not be received until *all* modules in the class have received packets $(\infty)$ on all of their input ports which receive packets from outside the class, and all modules at some time have ceased data operations. Thus the second test cannot be initiated until the first termination requirement for the class is satisfied. Each successive test cannot be initiated until the previous one has completed. This not only simplifies the termination operations, it limits the frequency with which tests can be initiated.

## Correctness of the Termination Operations

The addition of the termination operations to the simulation modules will not interfere with the simulation of the system, but they will cause the simulation to terminate if the actual system does. This is stated in the following theorem.

---

<u>Theorem</u> <u>2</u>. Correctness of Termination

a.) Suppose a simulation is performed in which the modules to be simulated obey the three requirements: functionality of output, monotonicity of output, and finite delay, and the simulation and coordination operations of each simulation module obey the three requirements: correct module simulation, correct ordering of output packets, and correct coordination, and furthermore the coordination operations of a simulation module cannot cause time packets (∞) to be sent out by the simulation module unless

$$\min_{1 \le k \le n} (tlast_k) = \infty.$$

Then the addition of termination operations to the simulation modules as described in Chapter 3 will not cause any of these requirements to be violated.

b.) If the actual system ever reaches a state in which no modules in the system will ever enter the firing mode unless more packets are received on the system input ports, then every simulation module in the simulation of this system will eventually produce time packets with value ∞ on all output ports, if all system input ports in the simulation receive time packets with value ∞.

---

The proof of this theorem is included in Appendix 2 and will be described here briefly. The termination operations for different connectivity classees are separate, hence we need only prove that the operations are correct for each class. Moreover, since the termination operations are designed not to interfere with the normal simulation and coordination operations, the only possible adverse effect of the termination operations is to terminate the simulation too soon. Thus, proving the first part of the theorem involves proving that the simulation modules in a class will not terminate until a test of the class suceeds, and that a test will suceed only if the termination conditions for the class are satisfied. In other words, if the termination control module T sends out (test.+) packets, then all K returning test packets will have value test.+ only if the termination conditions are satisfied. Proving that a test of a class will not overlook some simulation module which is not yet ready to terminate constitutes the most difficult part of the entire proof of correctness.

To prove the second part of the theorem, it must first be shown that a test of the class and a subsequent reset will eventually be completed, unless the termination conditions for the class are never satisfied. In other words, any time the termination control module sends out test or reset packets, it will eventually receive K test or reset packets, unless some simulation module $M_i$ never receives a time packet (∞) on some input port which is not in $from\_class_i$, or some actual module runs indefinitely. Thus, once the termination conditions for the class are satisfied, any previous test or reset operations will be completed, and a new test will be initiated. Furthermore the reset operations must cause all modules in the class to receive at least one (reset) packet before the new test packets are received. Finally, it must be shown that a test will suceed, once the termination conditions are satisfied.

## Conclusion

The relatively simple coordination operations of Chapter 3, which are designed to keep the simulation from deadlocking, created a much more difficult problem of terminating the simulation. The solution of this problem requires both compromising the modularity of design of the simulation modules to some degree and also adding termination operations which are more complex than the original coordination operations. This lack of modularity and greater complexity makes the correctness of the termination operations more difficult to prove than the correctness of the simulation and coordination operations.

However, the termination operations do satisfy the design goals for the simulation. The simulation remains a packet communication architecture system in which all communication is in the form of packets, the simulation modules are autonomous, and the design is time-independent. Furthermore, while the termination operations are more complex than the coordination operations, their implementation should not be particularly difficult, and they are efficient enough to have little effect on the speed of the simulation.

# Chapter 5

# Improving the Efficiency of the Simulation

## Introduction

The coordination algorithm of Chapter 3 is rather primitive in that the coordination operations of a simulation module make little use of the properties of the actual module, other than its minimum delay time *delay*. This leads to a simulation which requires a great deal of coordination information to be passed between simulation modules and which unnecessarily restricts the concurrency of the simulation.

Any modification to the coordination methods must preserve their desirable properties. The coordination operations should be simple enough to be easily incorporated in the simulation program for a module. The simulation should still be a packet communication architecture system, hence there should be no centralization of control or timing restrictions on the simulation modules or the communication links between them. Finally, the design should be modular - the coordination operations for a module should depend only on that module and not on the structure of the rest of the system.

In this chapter, two methods which can increase the efficiency under some conditions will be presented. These two particular modifications were chosen, because they are easy to implement and apply to many packet communication architecture systems. It will be shown that with either of these two modifications, the Correctness of Simulation Theorem, described in Chapter 3, will still apply.

## Modules which Compute Monotone Functions

Many of the packet communication architecture modules which have been designed to date compute *monotone functions* over their histories. That is, if the module produces an output history $HO_1$ when given the input history $HI_1$, and an output history $HO_2$ when started in the same initial state and presented with an input history $HI_2$, where

$$HI_1 \sqsubseteq HI_2,$$

then

$$HO_1 \sqsubseteq HO_2.$$

Modules which compute monotone functions over their histories are characterized by the property that the decision about which input packets are absorbed from each input port and used in a particular firing is independent of the arrival times of any input packets.

In particular, any *determinate* module computes a monotone function, where a determinate module [12,18] is a module for which the sequences of output packets sent from the output ports depend only on the sequences of input packets arriving at the input ports, and not on their arrival times. For example, the functional operator and switch modules of Chapter 1 are determinate modules.

One would expect many packet communication architecture modules to be determinate, since they embody the ultimate form of time-independent operation.

For example, all of the data flow actors of Dennis [5] have determinate behavior, so by the Closure Theorem of Determinate Systems of Patil [18], any module which implements a data flow program must be determinate. One important module which does not compute a monotone function over histories and therefore is not determinate is the arbiter module. The order in which packets are absorbed and subsequently sent out depends on the relative arrival times of the packets on each input port.

Other modules are nondeterminate, but do compute a monotone function over histories. For example, a system clock module which, when it receives a packet of the form (request_time), sends out a packet containing the time at which the request packet arrived, computes a monotone function over histories, but its output values depend on the times at which the input values were received.

## Simulation of Modules which Compute Monotone Functions

If a module computes a monotone function, then it can be safely fired in the simulation as soon as the necessary data packets have arrived at the input ports. There is no need to make sure that $tfire \leq \min_{1 \leq k \leq n} (tlast_k)$. Thus, the simulation module can use any of the input data packets, and not just those with time values less than or equal to $\min_{1 \leq k \leq n} (tlast_k)$.

For example, if the simulation module for an ADD module has received a packet (x,10) on input port 1, and a packet (y,20) on input port 2, then there is no need to wait until a packet with time ≥ 20 has been received on input port 1. Instead, the firing of the module at time 20 can be simulated right away, since any data packet received on input port 1 would not affect this firing.

As long as this revised firing rule does not cause any of the three requirements for the simulation module to be violated: correct module simulation, correct ordering of output packets, and correct coordination, the Correctness of Simulation Theorem presented in Appendix 1 will still hold. To show that this modification will not violate the correct module simulation requirement, suppose at some time a simulation module for a module which computes a monotone function has received an input history HSI', where HSI' $\subseteq$ HSI, the input simulation history which will ultimately be received. Then if all possible firings of the module on the data packets are simulated, and an output simulation history HSO' is produced, the effect of these activities will be to simulate the operation of the actual module as if it had received an input history HI', where

$$HI' = data(HSI').$$

We know that

$$HI' \subseteq HI,$$

where HI = data(HSI). Hence, since the module computes a monotone function,

$$HO' \subseteq HO,$$

where HO' is the actual module's output history in response to HI', and HO is the actual module's response to HI, when started in the same initial state. In simulating the actual module's operations on the history HI', a simulation history HSO' has been produced where

$$\mathrm{data(HSO')} = \mathrm{HO'} \sqsubseteq \mathrm{HO}.$$

The revised firing rules will not cause the module to fire prematurely. Thus, the first requirement, correct module simulation, will not be violated. Furthermore, this modification will not affect the rules for producing time packets. Thus, the other two requirements will still be valid: correct ordering of output packets and correct coordination. The Correctness of Simulation Theorem still applies.

This modification will improve the efficiency of the simulation by increasing the concurrency of module simulations. There is no need for a module which computes a monotone function to wait for time or data packets when sufficient data packets are already present. Furthermore, it actually simplifies coordination operations, since there is no longer any need to determine whether a module can be safely fired.

## Strengthening the Calculation of the Minimum Output Time

In the coordination algorithm of Chapter 3, *tout*, the earliest possible time at which the simulation could next send out a data packet, is calculated as

$$tout = \min_{1 \le k \le n} (tlast_k) + delay,$$

where $tlast_k$ is the time value of the last packet received on input port $i_k$. In other words, it was assumed that the firing of a module might be simulated as soon as any packet arrives on whichever input port $i_k$ currently has the lowest value of $tlast_k$. In many cases, however, the module would not be enabled to fire, even if such a packet were received. For example, if the simulation module for an ADD module has not received any data packets, and $tlast_1 = 100$, and $tlast_2 = 10$, then the firing of the module for any time less than or equal to 100 will never be simulated, even if a packet with time value 11 is received on input port 2. The coordination operations are overly cautious. They assume only something which is true for any module - if there are not sufficient packets for the module to fire, then the module cannot fire before the arrival of the next packet. If the coordination operations could take advantage of the firing requirements for a module, then it could often calculate values of *tout* which are higher than those obtained by the method of Chapter 3.

Any change in the method of calculating *tout*, will inevitably be more complex than the calculation

$$tout = \min_{1 \le k \le n} (tlast_k) + delay.$$

Hence, the strength of the calculation, that is the closeness to the maximum possible value, must be balanced with the simplicity of the calculation. The following method of calculating *tout* represents a particular compromise between strength and simplicity. It is very simple yet seems to be reasonably strong for many modules.

## Expressing the Firing Requirements

First, a method of specifying under what conditions a module might fire is required. For any module, a boolean-valued function F can be given which takes as arguments the values of $p_j$, $1 \le j \le n$, where $p_j$ is the number of packets present at input port $i_j$. If

$$F(p_1, p_2, \ldots, p_n) = \underline{\text{true}},$$

then the module might fire when $p_j$ packets are present at each input port $i_j$. If the value of the function is <u>false</u>, however, then regardless of the internal state of the module, the time, or any stochastic processes within the module, if each input port $i_j$ contains exactly $p_j$ input packets for all $j$, $1 \le j \le n$, and the module is in the wait mode, then the module cannot possibly enter the firing mode. Thus, as long as the value of the function is <u>false</u>, the module cannot produce any packets until more packets are received.

For example, an ADD module has a function

$$F_{ADD}(p_1,p_2) = (p_1 \ge 1) \wedge (p_2 \ge 1).$$

It cannot fire unless each each of the input ports contains at least one packet. The arbiter has a function

$$F_{Arb}(p_1,p_2) = (p_1 \ge 1) \vee (p_2 \ge 1).$$

It can fire if there is a packet on either input port. As a final example, if the behavior of the module is totally unpredictable, a function

$$F_{true}(p_1,p_2,\ldots,p_n) = \underline{true},$$

can always be used. This will apply even for modules which can sometimes fire without receiving any packets, since there are no conditions for which the value of the function is <u>false</u>, but the module can fire.

An equation for *tout* can be derived for a simulation module, if the equation for F of the corresponding actual module is expressed in the following form:

$$F(p_1,p_2,\ldots,p_n) = \begin{array}{l} \quad [(p_1\geq c_{11}) \wedge (p_2\geq c_{21}) \wedge \ldots \wedge (p_n\geq c_{n1})] \\ \vee\ [(p_1\geq c_{12}) \wedge (p_2\geq c_{22}) \wedge \ldots \wedge (p_n\geq c_{n2})] \\ \qquad\qquad\qquad\qquad \vdots \\ \vee\ [(p_1\geq c_{1q}) \wedge (p_2\geq c_{2q}) \wedge \ldots \wedge (p_n\geq c_{nq})], \end{array}$$

in which each $c_{kj}$ is some constant greater than or equal to zero. This form of the equation is called the *sum of products* form. Note that if $c_{kj} = 0$, then the predicate $(p_k\geq c_{kj})$ must have value <u>true</u>, thus these factors can be omitted from the equation. Equations with all factors of the form $(p_k\geq 0)$ removed are in *reduced* sum of products form. In the preceding examples, the functions $F_{ADD}$, $F_{Arb}$, and $F_{true}$ are expressed in reduced sum of products form.

Many functions cannot be expressed in this sum of products form. In fact, only those functions for which

$$F(p_1,p_2,\ldots,p_n) = \underline{\text{true}}$$

implies that for any values, $k_1,k_2,\ldots,k_n \geq 0$,

$$F(p_1+k_1,p_2+k_2,\ldots,p_n+k_n) = \underline{\text{true}},$$

can be expressed in this form. However, for any function F we can always find a "weaker" function F', such that if

$$F(p_1,p_2,\ldots,p_n) = \underline{\text{true}}$$

then

$$F'(p_1,p_2,\ldots,p_n) = \underline{\text{true}},$$

and an equation for F' can be expressed in sum of products form.

A sum of products equation for F can be translated into an equation for *tout* as follows:

$$tout = \mathrm{MAX}[ \min_{1 \le k \le n}(tlast_k) + delay : \min_{1 \le j \le q} ( \max_{1 \le k \le n} (t_{kc_{kj}})) + delay - \epsilon ],$$

where

$t_{kl}$ = the earliest possible time value of the $l$th packet on input port $i_k$
= the time value of the $l$th packet on $i_k$, if $l \le p_k$, or
= $tlast_k$, if $l > p_k$,

*delay* = the minimum delay time of the module, and

$\epsilon$ = any number greater than zero.

The second term of the equation

$$\min_{1 \le j \le q} (\max_{1 \le k \le n} (t_{kc_{kj}})) + delay - \epsilon,$$

represents the calulation of the minimum output time based on the function F. As will be proved shortly, for any value $t'$ such that

$$t' < t_0 = \min_{1 \le j \le q} (\max_{1 \le k \le n} (t_{kc_{kj}})),$$

if $p'_k$ is the number of packets on input port $i_k$ with time values less than or equal to $t'$, then

$$F(p'_1, p'_2, \ldots, p'_n) = \underline{\text{false}},$$

Hence, the module cannot possibly fire again before time $t_0$, and no data packets with time values less than or equal to $t_0$ + *delay* can be produced by the simulation module. Since all packets in the simulation must be sent from each output port in strictly increasing order, the term $\epsilon$ is required for *tout* to be strictly less than the time value of the next data packet.

If the calculation of *tout* were based only on the function F, it might be overly cautious. It is possible for the function F to have value true even when the module cannot possibly fire. In this case, a calculation of the minimum output time based on the equation for F would give a value which is too low.

Even if the function F has value true at some point in the simulation, if the data packets with time values less than or equal to $\min_{1 \le k \le n}(tlast_k)$ are not sufficient for the module to fire, then no data packets can be produced with time values less than or equal to $\min_{1 \le k \le n}(tlast_k) + delay$. Thus, the calculation of *tout* must take the maximum of the two predictions of the minimum output time - that based on the function F, and that based on the values of $tlast_k$.

For example, for the ADD module the equation is

$$tout = \mathrm{MAX}[\ \min(tlast_1, tlast_2)+delay\ ;\ \max(t_{11}, t_{21})+delay-\epsilon\ ].$$

For the arbiter, the equation is

$$tout = \mathrm{MAX}[\ \min(tlast_1, tlast_2)+delay\ ;\ \min(t_{11}, t_{21})+delay-\epsilon\ ],$$
$$= \min(tlast_1, tlast_2) + delay.$$

This equation degenerates to the original equation for *tout*. Finally, for the function $F_{true}$ the equation is

$$tout = \mathrm{MAX}[\ \min_{1 \le k \le n}(tlast_k)+delay\ ;\ 0+delay-\epsilon\ ]$$
$$= \min_{1 \le k \le n}(tlast_k) + delay.$$

This equation also degenerates to the original equation for *tout*.

## Correctness of the Calculation

this modified method of calculating *tout* will not cause the simulation to violate any of the three requirements: correct module simulation, correct ordering of output packets, or correct coordination. Hence, the Correctness of Simulation Theorem given in Appendix 2 will still apply. Clearly the correct module simulation requirement will still hold, since this modification will not affect the data packets produced by the module in the simulation.

As for the correct ordering of output packets requirement, a time packet will not be sent out from output port $o_j$ with time value less than or equal to *tlast-out*$_j$, since this is checked for by the simulation module. The only danger is that a time packet with value *tout* might be sent out, and later a data packet with time less than or equal to *tout* is sent out. The original proof shows this cannot happen for $tout = \min_{1\le k\le n}(tlast_k) + delay$, hence the problem can only occur if

$$tout = \min_{1\le j\le q}\left(\max_{1\le k\le n}(t_{kc_{kj}})\right) + delay - \epsilon.$$

The claim, however, is that for any value $t'$ such that

$$t' < t_0 = \min_{1\le j\le q}\left(\max_{1\le k\le n}(t_{kc_{kj}})\right),$$

if $p'_k$ is the number of packets on input port $i_k$ with time values less than or equal to $t'$, then

$$F(p'_1, p'_2, \ldots, p'_n) = \underline{\text{false}}.$$

Hence the module cannot fire again in the simulation at any time, $t' < t_0$. To show this, look at any $t_{kc_{kj}}$ for which

$$t_{kc_{kj}} = \max(t_{1c_{1j}}, t_{2c_{2j}}, \ldots, t_{nc_{nj}}).$$

By our assumption about $t'$, and from the equation for $t_0$

$$t' < t_0 \le t_{kc_{kj}},$$

and $t_{kc_{kj}}$ by definition is the earliest possible time value of the $c_{kj}$th data packet on input port $i_k$. Thus, $p'_k < c_{kj}$, which implies that the predicate

$$(p'_k \ge c_{kj}) = \underline{\text{false}}, \text{ for any } j,\ 1\le j\le q.$$

This means that for any $j$, the product term

$$(p'_1 \ge c_{1j}) \wedge (p'_2 \ge c_{2j}) \wedge \ldots \wedge (p'_n \ge c_{nj}) = \underline{\text{false}}.$$

Therefore, F, which is the sum of these product terms must have value <u>false</u>.

No firing of the module before time

$$t_0 = \min_{1 \le j \le q} \left( \max_{1 \le k \le n} (t_{kc_{kj}}) \right),$$

can be simulated, hence no data packets can be produced with time values $\le t_0$ + *delay* can be produced. If

$$tout = t_0 + delay - \epsilon,$$

and $\epsilon > 0$, the correct ordering of output packets requirement will not be violated.

Finally, the correct coordination requirement will not be violated, since

$$tout \ge \min_{1 \le k \le n} (tlast_k) + delay > \min_{1 \le k \le n} (tlast_k),$$

unless $\min_{1 \le k \le n} (tlast_k) = \infty$. Thus, the Correctness of Simulation Theorem of Appendix 1 will still hold for this revised calculation of *tout*.

## Compatibility with the Termination Operations

One difficulty caused by this revised calculation of *tout* is that the calculation might cause a simulation module to produce time packets with value $\infty$ before time packets with value $\infty$ have arrived on all input ports. This could interfere with the termination operations for the connectivity class. If some other simulation module receives one of these time packets, it will assume that the most recent test succeeded and will send out time packets $(\infty)$ from all output ports, which might not be valid.

One way to prevent this problem would be to require that no simulation module send out $(\infty)$ packets, until all input ports have received $(\infty)$ packets. Instead, when $tout = \infty$, it would send out time packets $(t)$ where $t$ is some "large" number. This seems rather awkward, but it will prevent the *tout* calculations from interfering with the termination operations.

## Features of the Calculation

This calculation of the minimum output time uses information which is already available to the simulation module, namely the time values of each data packet at the input ports and the values of $tlast_k$. No attempt is made to predict the time value of the $l$th packet if $p_k < l$, except that it is greater than $tlast_k$. This avoids passing more coordination information between simulation modules, or requiring knowledge of the timing details of the other simulation modules.

This calculation of *tout* is reasonably simple, in fact hardly more complex than the original calculation. One reason for this simplicity is that it ignores much of the information which is available to the simulation module. For example, the data values of the input packets are not considered, nor is the state or time of the module. Under some circumstances this will lead to a weaker calculation of *tout* than might be possible. If the conditions under which a particular module can fire depend heavily on these factors, it would be worthwhile to take these factors into account when calculating *tout*.

This method of calculating *tout* will increase the efficiency of the simulation in two ways. First, it will decrease the number of time packets sent between simulation modules. Not only will the difference between successive time values tend to be greater, the need to send time values around loops a number of times just to fire a module once can be reduced. For example, suppose the module $M_1$ of Figure 5.1 obeys the function

$$F(p_1,p_2) = (p_1 \geq 1) \wedge (p_2 \geq 1).$$

Using the original method of calculating *tout*, *tout* = min(10,100) + 2 = 12. Thus a time packet (12) would be sent to $M_2$, which would send back a time packet (13) and so on, until after $M_2$ has sent 30 time packets, it would finally receive the packet (100) and the firing at time 100 could be simulated. If instead we use the calculation

$$tout = \mathrm{MAX}[\ \min(10,100)+2\ ;\ \max(10,100)+2-0.001\ ] = 101.999,$$

the time packet (101.999) could be sent to $M_2$, which would send back (102.999), and the firing of the module could be simulated. Thus, the reduction in the number of packets sent during the simulation can be very large.

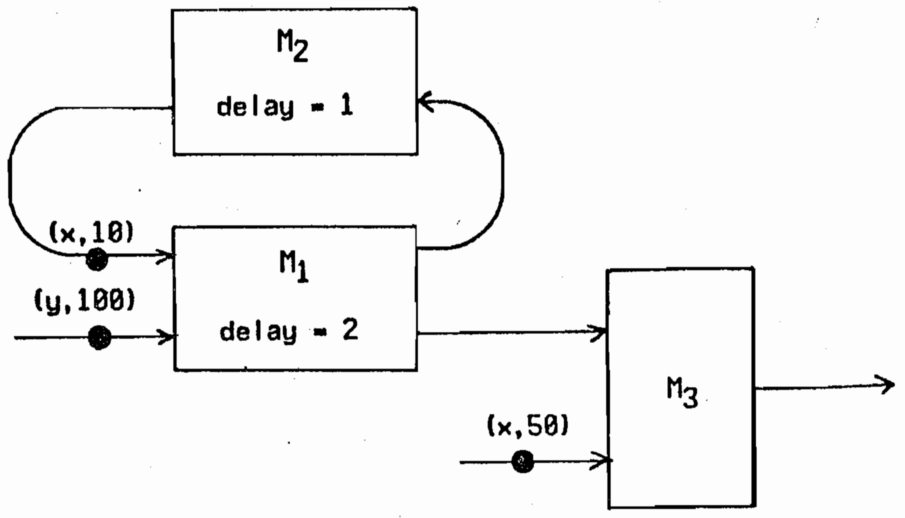


*Figure 5.1* - System which can be Simulated More Efficiently with Stronger *tout* Calculations.

The second improvement in the efficiency comes in the form of increased concurrency of the simulation. In the previous example, $M_1$ would not need to wait for time packets to cycle through the loop 30 times before firing. Furthermore, if there were some module $M_3$ connected to output port $o_2$ of $M_1$ which is waiting for a time packet with time greater than or equal to 50 from $M_1$, it would receive this packet much sooner. By reducing the time spent sending and waiting for time packets, the simulation modules can spend a proportionately larger amount of time simulating the data operations of the modules. This would increase the concurrency of the module simulations.

## Conclusion

These two modifications were chosen, because they can be easily implemented and make use of properties which are expected to be common in packet communication architecture systems. Other modifications could improve the efficiency of the simulation in other cases without compromising the desirable properties of the original method.

# Chapter 6

# Conclusion

## Insights and Afterthoughts

As has been demonstrated here, it is indeed possible for the simulation of a packet communication architecture system to itself fulfill the design philosophies of packet communication architecture. The modularity and time-independence of the simulation allows it to be performed by virtually any computer system which supports intercommunicating processes. Furthermore, the operations which must be performed for each module in the system are reasonably simple and therefore can be executed by small processors such as microprocessors.

The methods which have been developed here are very general as well. Few restrictions are placed on either the characteristics of the modules in the system or on how these modules are interconnected. Moreover, the methods are provably correct, which is an important feature for any asynchronous, parallel computation, due to the numerous and often subtle difficulties which are encountered in the design of such systems.

The coordination and termination operations are simple enough to use only a small fraction of the simulation module's processing time. However, it is difficult to estimate what fraction of the processing time will be spent waiting for the necessary time or data packets. This will depend a great deal on the structure of the simulation facility and on the system to be simulated. Thus, it is difficult to estimate the efficiency of the simulation, that is what fraction of the processing time will be spent simulating the activities of the modules. However, considering the low efficiency of a simulation on a sequential computer system, the efficiency of the parallel simulation seems quite reasonable by comparison.

Perhaps the fundamental philosophy which is expressed in this work is that a certain amount of *overhead*, that is computation whose only purpose is to maintain proper operation of the system, is needed for all but a limited class of computer systems. This fact was accepted long ago by designers of traditional computer systems. For example, many of the functions performed by an operating system are overhead. Such operations as memory paging and resource scheduling are incidental to the execution of a user's program. Similarly, the coordination and termination operations of the simulation modules are incidental to the simulation of the activities of the actual system. In a distributed computation, the increase in the system load caused by the overhead operations appears in two forms: as added computations for the components of the system, and as special control information sent between the components.

These overhead operations are acceptable if they are kept to a minimum and are designed in such a way that they both preserve the design goals of the system and remain invisible to the user of the system. For example, the amount of overhead in the simulation is reasonably small, the principles of packet communication architecture are preserved, and the overhead operations are invisible to people performing simulations.

The design of overhead computations for parallel systems is still in a rather primitive state. Other parallel computer systems, such as Illiac IV [3], are structured in such a way that the amount of overhead operartions is minimized. These systems contain central controllers which tightly control the operations of the components, thereby avoiding the need for the processors to communicate their status with one another. Because of the rigid control structure, however, it is difficult for the user to program such a system to run efficiently. These systems are suitable only for applications in which the structure of the algorithm closely matches the structure of the system.

Packet communication architecture systems, with their decentralized control and time-independent operation are potentially much more flexible and general purpose than other parallel systems. However, along with this increased capability comes a need for the components of the system to keep their activities coordinated properly. The design of overhead operations for these systems requires an approach which is totally different from those used in designing traditional systems. The overhead computations incorporated in each component of the system can utilize only a limited amount of information about the rest of the system. For example, the only information about the status of the rest of the system available to the coordination and termination operations of each simulation module is in the form of time and test packets received at the input ports. Overhead operations which can be "modularized" in this fashion seem rather foreign, partly because they have no locus of control. Instead, the operations take place in many locations simultaneously.

Furthermore, while one component of the system is performing operations, the state of the rest of the system can be changing. The overhead operations must be designed to operate correctly, despite a continuously changing system state. As a result, one cannot fully understand how the operations work by focusing on one component at a time. The system must be viewed as a whole to see how the operations work. For example, the termination operations performed by each simulation module make little sense when viewed individually, but they fit together into a computation which will detect when the simulation can be terminated.

To date, no general techniques for designing the overhead operations in packet communication architecture systems have been developed. Instead, they have been designed on a case-by-case basis, taking advantages of special properties of the system. For example, the design here takes advantage of the fact that the sole purpose of a simulation is to model the behavior of some other system. If the actual system contains deadlocks or other malfunctions, the simulation should model these deadlocks and malfunctions. The burden of designing a system free of errors is left up to the system designer. In the future, however, general techniques should evolve which make the overhead operations both easier to design and understand.

## Suggestions for Further Research

There are two directions in which further research can build upon the work which has been presented here. First, more work is required before packet communication architecture systems can be simulated. In particular, a means of programming the simulation modules is needed. Ideally, the user of a simulation facility should be able to specify the operations of the components of the actual system in a high-level language, such as the Architecture Description Language of Leung, *et al* [14]. These specifications would then be translated into programs for the simulation modules by an ADL compiler. The user should not be concerned with the coordination and termination operations, nor with the details of the module activity simulation. Fortunately, the coordination and termination operations are simple and uniform enough that they will not increase the complexity of this translation greatly. The major difficulty is the design of a language which allows the specification of a wide variety of systems in a concise and understandable form, but can be translated into programs for the simulation modules. With the increasing interest in parallel, asynchronous computing systems, a convenient and efficient means of simulating them will be required to determine the best designs.

The other potential direction for further research is to apply some of the techniques and insights which have been developed here to other areas. One direct application would be to the simulation of systems which are not strictly packet communication architecture systems. Some systems which are commonly simulated, such as air traffic control models, have the essential properties of packet communication architecture design. That is, the system can be subdivided into a number of components which operate independently and communicate with each other only in a limited and well-defined manner. For example, an air traffic control model can be subdivided into geographic regions.

The activities within each region occur simultaneously and independently. The only communication is between neighboring regions, and the only way they communicate is by changing the boundary conditions. The simulation techniques which have been developed here can be applied directly to such systems. This will lead to a highly parallel simulation which can be executed by a relatively simple network of computers. For the air traffic control model, one can envision a "grid" of processors, in which each processor simulates the activities within one geographic region. The simulation of an air traffic control model on a network of processors has been studied in some detail by Thomas and Henderson [22]. In their system, different geographical regions of a hypothetical airspace are simulated on different Arpanet processors. The simulator for one region sends a message to the simulator for an adjacent region when a plane crosses from the first region into the second. To maintain proper time synchronization, one of the simulators maintains a global time clock and broadcasts the simulation time to the other simulators at regular intervals. In their description of the system, the authors note that a distributed approach to time synchronization would be preferable, since this centralized approach tightly binds the simulators to the global clock. It seems that coordination operations along the lines of those presented in Chapter 3 could provide the necessary synchronization. Each simulator would send a time packet to the simulator for each adjacent region indicating the earliest possible simulation time at which a plane could possibly cross from the first region into the next. In this way, the simulation can proceed without any centralized control or real-time constraints on the simulators.

Moving beyond the field of simulation, there are other areas to which these techniques and insights can be applied. The problems of deadlock and nontermination which were dealt with here occur frequently in parallel, asynchronous systems. The concept of adding overhead operations to a system to prevent these problems can be applied to other systems. For example, the author [4] has identified a deadlock which can occur when the data flow language of Weng [23] is extended to include both cycles and nondeterminacy. This deadlock occurs after all computation by the program is completed, but the program fails to recognize that it is able to terminate. This deadlock can be avoided by adding more data flow actors to the program to perform the necessary overhead operations and terminate the program. In fact, these overhead computations are very similar to the termination operations of the simulation modules.

To design the overhead operations for a wider class of parallel, asynchronous systems, however, more general techniques will be required. Ideally, a programmer should be able to specify a program in a high-level language which will then be compiled into a number of separate module programs which include all of the needed overhead operations. These programs could then be loaded into the modules of a packet communication architecture system, and the system would then execute the program in a highly parallel fashion. Translating high-level languages which include such features as data structures and recursive procedure calls into individual module programs will pose many difficulties.

Thus, while the focus of this work was on simulating a particular type of computer system in a particular manner, some of the techniques and concepts which were developed here have much broader areas of application.

## Bibliography

[1] Aho, A. V., J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass. (1974).

[2] Anderson, G. A., and E. D. Jensen, "Computer Interconnection Structures: Taxonomy, Characteristics, and Examples", Computing Surveys, Volume 7, Number 4, (December 1975), pp. 197-212.

[3] Barnes, G. H., *et al*, "The Illiac IV Computer," IEEE Transactions on Computers, C-17, Vol. 8, IEEE, New York (August 1968), pp. 746-757.

[4] Bryant, R. E., "Nondeterminate Stream-Oriented Computations by Cyclic Data Flow Systems," Unpublished paper, Laboratory for Computer Science, MIT, Cambridge, Mass. (December 1976).

[5] Dennis, J. B., First Version of a Data Flow Procedure Language, Technical Memorandum TM-61, Laboratory for Computer Science, MIT, Cambridge, Mass. (May 1975).

[6] Dennis, J. B., "Packet Communication Architecture," Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, IEEE, New York (August 1975), pp. 224-229.

[7] Dennis, J. B., and D. P. Misunas, "A Computer Architecture for Highly Parallel Signal Processing," Proceedings of the ACM 1974 National Conference, ACM, New York (November, 1974), pp. 402-409.

[8] Dennis, J. B., and D. P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," Proceedings of the Second Annual Symposium on Computer Architecture, IEEE, New York (January 1975), pp. 126-132.

[9] Dennis, J. B., D. P. Misunas, and C. K. Leung, A Highly Parallel Processor Using a Data Flow Machine Language, Computation Structures Group Memo 134, Laboratory for Computer Science, MIT, Cambridge, Mass. (January 1977).

[10] Ellis, D., private communication.

[11] Farber, D. J., *et al*, "The Distributed Computing System," Proceedings Seventh Annual IEEE Computer Society International Conference, IEEE, New York (February 1973), pp. 31-34.

[12] Kahn, G., "A Preliminary Theory for Parallel Programs," Internal Memo, Institut de Rech. d'Informatique et d'Automatique, Rocquencourt, France (1973).

[13] Kay, I. M., T. M. Kisko, and D. E. Van Houweleng, "GPSS/Simscript - The

Dominant Simulation Languages," Proceedings of the Eighth Annual Simulation Symposium, IEEE, New York (1975) pp. 141-154.

[14] Leung, C. K., D. P. Misunas, A. Neczwid, and J. B. Dennis, "A Computer Simulation Facility for Packet Communication Architecture," Proceedings of the Third Annual Symposium on Computer Architecture, IEEE, New York (1975), pp. 58-63.

[15] Metcalfe, R. M., Packet Communication, Technical Report TR-114, Laboratory for Computer Science, MIT, Cambridge, Mass. (December, 1973).

[16] Organick, E. I., Computer System Organization: the B5700, B6700 Series, Academic Press, New York (1973).

[17] Paley, H., and P. M. Weichsel, A First Course in Abstract Algebra, Holt, Rinehart, and Winston, Inc., New York (1966).

[18] Patil, S. S., "Closure Property of Interconnected Systems," Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, New York (1970), pp. 107-116.

[19] Rowe, L. A., The Distributed Computing Operating System, Technical Report Number 66, Department of Information and Computer Science, University of California at Irvine, Irvine, Calif. (June, 1975).

[20] Rumbaugh, J. E., A Parallel, Asynchronous Computer Architecture for Data Flow Programs, Technical Report TR-150, Laboratory for Computer Science, MIT, Cambridge, Mass. (May 1975).

[21] Swan, R. J., S. H. Fuller, and D. P. Sieworek, "Cm*: A Modular, Multi-Microprocessor," A Collection of Papers on Cm*: A Modular, Multi-Microprocessor, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA. (February 1977).

[22] Thomas, R. H., and D. A. Henderson, "McRoss - A Multi-Computer Programming System," 1972 Spring Joint Computer Conference, AFIPS, Montvale, N. J. (1972), pp. 282-293.

[23] Weng, K. S., Stream-Oriented Computations in Recursive Data Flow Schemas, Technical Memo, TM-68, Laboratory for Computer Science, MIT, Cambridge, Mass. (October 1975).

# Appendix 1

## Correctness of the System Simulation

The following proof shows that the simulation operations of Chapter 2, combined with the coordination operations of Chapter 3 will give a simulation which accurately models the actual system.

Before proceeding with the proof, some additional notation is needed. For an input port $i_k$ of a simulation module, the value of $tlast_k$ is the last time value received on that input port. Thus, for an input port simulation history, we can define a function Tlast where $\text{Tlast}(\text{hsi}_k)$ equals the minimum value of $t$, $0 \le t \le \infty$, such that $\text{hsi}_k(t) = \text{hsi}_k$. Similarly, for an output port $o_r$ of a module, $tlast\text{-}out_r$ equals the last time value sent from the port. Thus, a function Tlast-out can be defined for output port simulation histories, where $\text{Tlast-out}(\text{hso}_r)$ equals the minimum value of $t$, $0 \le t \le \infty$, such that $\text{hso}_r(t) = \text{hso}_r$.

Finally, for a module input simulation history HSI the function Tfinal is defined as:

$$\text{Tfinal}(\text{HSI}) = \min_{1 \le k \le n} (\text{Tlast}(\text{hsi}_k)),$$

where

$$\text{HSI} = \langle \text{hsi}_1, \text{hsi}_2, \ . \ . \ . \ , \text{hsi}_n \rangle.$$

This function can be applied to system input simulation histories as well.

## Requirements of the Simulation

The correctness proof will apply to simulations which fulfill the following six conditions. First, there are three conditions on the modules to be simulated:

1.) **Functionality of Output:** The output history and final state of a module depend only on the initial state of the module and the input history.

2.) **Monotonicity of Output:** The output of a module at time $t$ cannot be affected by input received after time $t$.

3.) **Finite Delay:** The output of a module at time $t$ cannot be affected by input received *at* time $t$. In other words, there must be a finite delay between the receipt of an input packet and the production of an output packet which depends on this input packet.

If a module satisfies all three of these requirements, then its output history up to and including time $t$ must be a function of its initial state and its input history up to but not including time $t$. This can be specified more formally in terms of histories. Suppose for two operations of a module, the module produces an output history HO when it starts in initial state $S_0$ and receives the input history HI, and it produces an output history HO' when started in the same initial state $S_0$ and given the input history HI'. Then for any value of $t$ such that

$$\mathrm{HI}(t-\delta) = \mathrm{HI}'(t-\delta), \text{ for all } \delta>0,$$

the two output histories must be identical through time $t$, that is

$$\mathrm{HO}(t) = \mathrm{HO}'(t).$$

The following conditions will be required for each simulation module in the system:

1.) **Correct Module Simulation:** The simulation of a module must produce the same values as the actual module would under the same circumstances. That is, suppose the simulation of a module produces a simulation history HSO when it starts in initial state $S_0$ and receives input simulation history HSI, where all of the data and time packets arriving at each input port have strictly increasing time values. Let

$$tfinal = \text{Tfinal}(\text{HSI}),$$

That is, *tfinal* is the smallest of all the final time values received by the input ports of the simulation module. Then

$$\text{data}(\text{HSO}(tfinal)) = \text{HO}(tfinal),$$

where HO is the output history of the actual module when it starts in the same initial state $S_0$ and receives the input history HI = data(HSI). Furthermore, if *tfinal* = ∞ (all input ports to the module receive time packets with value ∞), then the final state of the simulation of the module $S_f$ will be the same as the final state of the actual module.

2.) Correct Ordering of Output Packets: If the packets arriving at each input port of a module in the simulation have strictly increasing time values, then the output packets sent from each output port of the module in the simulation will have strictly increasing time values.

3.) Correct Coordination: Each output port of a module in the simulation will eventually produce a time or data packet with time value *greater* than the minimum time value of the final packets received at the input ports, or else the output port will produce a time packet (∞). In other words, suppose a module in the simulation receives an input simulation history HSI and produces an output simulation history HSO. Then for any output port $o_r$ of the module either

$$\text{Tlast-out}(\text{hso}_r) > \text{Tfinal}(\text{HSI}),$$

or

$$\text{Tlast-out}(\text{hso}_r) = \infty.$$

The simulation and coordination operations (without the termination operations) presented in Chapters 2 and 3, satisfy all six of these requirements, as long as the modules to be simulated satisfy the first three requirements. First, the simulation operations developed in Chapter 2 will guarantee that the correct module simulation requirement is satisfied. To see this, suppose at some point in the simulation, a simulation module has received a simulation history HSI' where HSI' $\sqsubseteq$ HSI (the ultimate simulation history which will be received by the simulation module.) Assuming packets arrive at each input port with strictly increasing time values, then if

$$tmin = \mathrm{Tfinal}(\mathrm{HSI}') = \min_{1 \le k \le n} (tlast_k),$$

no new packets with time less than or equal to *tmin* will be received on any input port. By the firing rules for the simulation, the firing of the module at time *tfire* cannot be simulated, unless $tfire \le tmin$. Thus, when the firing of the module at time *tfire* is simulated the simulation history $\mathrm{HSI}(tfire)$ has been received. Assuming the simulation correctly simulates the firing of the module, the proper output packets will be produced. Furthermore, once the simulation module has received the entire input simulation history HSI with

$$tfinal = \mathrm{Tfinal}(\mathrm{HSI}),$$

the firing of the module for all values of $tfire \le tfinal$ will be simulated. Hence, all output packets with time values less than or equal to *tfinal* will be produced in response to this input simulation history, thereby guaranteeing that

$$\mathrm{data}(\mathrm{HSO}(tfinal)) = \mathrm{HO}(tfinal).$$

Thus the simulation will satisfy the correct module simulation requirement.

The second requirement, correct ordering of output packets, is met as long as the input packets to the simulation module are correctly ordered. That is, if an output port $o_j$ of the simulation module first produces a packet $p_1$ and then a packet $p_2$ then $t_1$, the time value in $p_1$, must be less than $t_2$, the time value in $p_2$. To show this, four cases must be considered:

1. $p_1$ and $p_2$ are both time packets.
Then $p_2$ would be sent out only if $t_2 > tlast\text{-}out_j = t_1$.

2. $p_1$ is a data packet and $p_2$ is a time packet.
As in case 1, $p_2$ would be sent only if $t_2 > tlast\text{-}out_j = t_1$.

3. $p_1$ and $p_2$ are both data packets.
Assuming the simulation module satisfies the correct module simulation requirement, data packets will always be produced in the proper order.

4. $p_1$ is a time packet and $p_2$ is a data packet.
$p_1$ was produced with a time value $t_1 = tmin + delay$ only if the module could not possibly fire before or at time $tmin$. The actual module always has a delay time greater than or equal to $delay$ between firing and producing output packets, hence the simulation module could not send out a data packet $p_2$ with time $t_2 \le t_1$ from the output port after $p_1$ has been sent.

For each of these four cases, the simulation will satisfy the correct ordering of output packets requirements.

The coordination operations also satisfy the correct coordination requirement. If the simulation module receives an input simulation history HSI with

$$tfinal = \mathrm{Tfinal(HSI)},$$

then after all output data packets have been produced, it will send out time packets with value

$$tout = tfinal + delay,$$

from all output ports for which $tout > tlast\text{-}out_j$. Since $delay$ is greater than zero, either $tout > tfinal$, or $tout = tfinal = \infty$. Hence, after the last time and data packets have been sent from each output port $o_j$, either

$$tlast\text{-}out_j \ge tout > tfinal,$$

or

$$tlast\text{-}out_j = tout = tfinal = \infty.$$

Thus, the correct coordination requirement will be satisfied.

A proof can now be given which shows that if the modules to be simulated satisfy their three requirements, and the simulations of these modules satisfy their three requirements, then when these simulation modules are interconnected, the simulation will accurately model the entire system.

---

__Theorem 1.__ Correctness of Simulation.

Suppose a simulation has the following properties:

1.) The modules to be simulated satisfy the monoticity of output, finite delay, and functionality of output requirements.

2.) The simulation of each module satisfies the correct module simulation, correct ordering of output packets, and correct coordination requirements.

3.) All communication links between simulation modules operate properly. In other words, if input port $i_k$ is connected to output port $o_r$ then $\mathrm{hsi}_k = \mathrm{hso}_r$.

4.) The simulation receives a system input simulation history SI and the sequence of time values received at each system input port is strictly increasing.

Let *tfinal* = Tfinal(SI), that is *tfinal* equals the smallest final time value received by any of the system input ports during the simulation. Then the simulation module for any module $M_j$ will produce a module output simulation history $\mathrm{HSO}_j$ such that

$$\mathrm{data}(\mathrm{HSO}_j(\mathit{tfinal})) = \mathrm{HO}_j(\mathit{tfinal}),$$

where $\mathrm{HO}_j$ would be the output history of the corresponding module in the actual system under following conditions:

1.) All modules in the actual system are started in the same initial state as the corresponding simulation modules.

2.) The actual system receives the system input history I, where
$$\mathrm{I} = \mathrm{data}(\mathrm{SI}).$$

Furthermore, if *tfinal* = ∞, the final state of each simulation module which terminates will equal the final state of the corresponding module in the actual system.

---

Before the major part of the theorem can be proved, two lemmas are needed.

---

__Lemma 1.1.__ Correct Ordering of All Packets

If the simulation of each module satisfies the correct ordering of output packets requirement, the communication links between the simulation modules operate correctly, and the packets arrive at each system input port with strictly increasing time values, then every output port of every simulation module will produce packets with strictly increasing time values.

---

__Proof of Lemma 1.1__

The proof will follow by induction on the sequence of packets which an observer would see if he were to simultaneously observe the output ports of every simulation module. This sequence would be of the form $p_1, p_2, \ldots, p_j, \ldots$ where $p_j$ is the $j$th packet observed. In any physical system, no two packets could appear at the *exact* same time, so the packets will be totally ordered in time. The sequence of packets sent from each output port is countable, and there are a finite number of output ports in the system, hence the sequence $p_1, p_2, \ldots$ must be countable. This allows us to perform induction on the sequence.

__Basis:__ Initially, no output ports have produced any packets, thus no ordering constraints have been violated.

__Induction:__ Assume the observer has seen the sequence $p_1, p_2, \ldots, p_l$ and up to this point, all output ports have produced packets with strictly increasing time values. Then, by the first-in, first-out property of the communication links, all input ports connected to these output ports have received packets with strictly increasing time values. Furthermore, all system input ports have received packets with strictly increasing time values. Hence, whichever module produces packet $p_{l+1}$ must have received input packets at each input port with strictly increasing time values up to this point. Since this simulation module satisfies the correct ordering of output packets requirement, the time value of $p_{l+1}$ must be greater than the time values of all packets which have been sent from this output port previously.

Thus, by induction, no packet in the sequence $p_1, p_2, \ldots$ can violate the ordering requirements for each output port.

---

<u>Lemma</u> <u>1.2.</u> Monotonicity of Simulation Output.

If a module satisfies the monoticity of output, finite delay, and functionality of output requirements, and the corresponding simulation module satisfies the correct module simulation requirement, then the output data packets produced by a module in the simulation with time values less than or equal to $t$ will depend only on the initial state and the input data packets received with time less than $t$. More precisely, suppose

$$\mathrm{data}(\mathrm{HSI}(t-\delta)) = \mathrm{HI}(t-\delta), \text{ for all } \delta>0,$$

and

$$t \leq \mathrm{Tfinal}(\mathrm{HSI}).$$

Then, if the actual module and the simulation module both start in the same initial state $S_0$

$$\mathrm{data}(\mathrm{HSO}(t)) = \mathrm{HO}(t),$$

where HSO is the output simulation history of the simulation module after receiving HSI, and HO is the output simulation history of the actual module after receiving HI.

---

The idea behind this lemma is that the simulation *can* and *will* produce the output simulation history $\mathrm{HSO}(t)$, once the input simulation history $\mathrm{HSI}(t-)$

has been received. That it *can* produce the output simulation history up to time $t$ is guaranteed by the three requirements on the module. That it *will* is guaranteed by the correct module simulation requirement. In order for the simulation module to realize it has received the entire input simulation history up to time $t$ it may require packets with time values greater than or equal to $t$, as is stated in the condition $t \leq \mathrm{Tfinal(HSI)}$. The simulation, however, will only use the packets with time values less than $t$ in calculating the output values with time values less than or equal to $t$.

<u>Proof</u> <u>of</u> <u>Lemma</u> <u>1.2</u>:

Let HI' = data(HSI), and let HO' equal the output history of the actual module when it starts in state $S_0$ and receives the input history HI'. Then by the statement of the lemma,

$$\mathrm{HI}(t-\delta) = \mathrm{data(HSI}(t-\delta)) = \mathrm{HI'}(t-\delta), \text{ for all } \delta>0.$$

Hence, by the three requirements for the actual module

$$\mathrm{HO'}(t) = \mathrm{HO}(t).$$

Furthermore, by the correct module simulation requirement, if *tfinal* = Tfinal(HSO), then

$$\mathrm{data(HSO}(tfinal)) = \mathrm{HO'}(tfinal).$$

By the statement of the lemma, $t \leq tfinal$, therefore

$$\mathrm{data(HSO}(t)) = \mathrm{HO'}(t).$$

Thus

$$\mathrm{data(HSO}(t)) = \mathrm{HO'}(t) = \mathrm{HO}(t).$$

This lemma will allow us to look only at the input data packets with time values less than $t$, when trying to prove the correctness of the simulation up to and including time $t$.

<u>Proof of Theorem 1</u>.

The main theorem will be proved by induction on the sequence of time values

$$t_0, t_1, t_2, \ldots, t_l, \ldots ,$$

where $t_0 = 0$, and

$$t_0 < t_1 < \ldots < t_l < \ldots \leq \infty,$$

and each time value $t_l$, $l > 0$, is contained in some actual or simulation history for the system. That is, $t_l$ is contained in one of the following histories: I, the system input history to the actual system; $\mathrm{HO}_j$, the output history of some module in the system $M_j$; SI, the system input simulation history; or $\mathrm{HSO}_j$, the output simulation history for some module $M_j$. As mentioned in Chapter 2, the history and simulation history for any port must be a countable sequence. Since there are only finitely many input and output ports in the system, only countably many time values can appear in all of the histories. Thus, the sequence $t_0, t_1, \ldots, t_l, \ldots$ must be countable, which allows us to perform induction on it.

<u>Induction Hypothesis</u>

For any $t_l \in t_0, t_1, \ldots, t_l, \ldots$, such that $t_l \leq$ *tfinal*:

a.) $\mathrm{data}(\mathrm{HSO}_j(t_l)) = \mathrm{HO}_j(t_l)$, for all modules $M_j$, *and*

b.) Either $t_l = \infty$, or for any output port $o_r$
$$\mathrm{hso}_r(t_l) \sqsubset \mathrm{hso}_r.$$

That is, the simulation will be correct through time $t_l$, and all output ports in the simulation will produce some packet with time value greater than $t_l$, unless $t_l = \infty$.

<u>Basis</u>: $l = 0$.

a.) Initially, $\mathrm{HSO}_j(0) = \mathrm{HO}_j(0)$ = the empty history, for any module $M_j$.

b.) Initially, $\mathrm{HSI}_j(0) = \mathrm{HI}_j(0)$ = the empty history. Hence, $\mathrm{Tfinal}(\mathrm{HSI}_j(0)) = 0$ for any module $M_j$. By the correct coordination requirement, for any output port $o_r$ of module $M_j$

$$\mathit{tlast\text{-}out}_r > \mathrm{Tfinal}(\mathrm{HSI}_j(0)) = 0.$$

Thus, $\mathrm{hso}_r(0) \sqsubset \mathrm{hso}_r$, for any output port in the system.

<u>Induction</u>: Assume true for $l$, where $t_l < \mathit{tfinal}$, prove true for $l$+1.

a.) The Monoticity of Simulation Output Lemma which has just been proved will be applied to show that $\mathrm{data}(\mathrm{HSO}_j(t_{l+1})) = \mathrm{HO}_j(t_{l+1})$. By the induction assumption

$$\mathrm{data}(\mathrm{HSO}_j(t_l)) = \mathrm{HO}_j(t_l).$$

for all modules $M_j$ in the system. Furthermore, by the statement of the theorem,

$$\mathrm{data}(\mathrm{SI}) = \mathrm{I}.$$

Therefore, since all communication channels in the simulation operate properly,

$$\mathrm{data}(\mathrm{HSI}_j(t_l)) = \mathrm{HI}_j(t_l),$$

for all simulation modules $M_j$. Since no packets are produced with time $t$ such that $t_l < t < t_{l+1}$,

$$\mathrm{data}(\mathrm{HSI}_j(t_{l+1}-\delta)) = \mathrm{HI}_j(t_{l+1}-\delta), \text{ for all } \delta>0.$$

Next, by part b). of the induction assumption $\mathrm{hso}_r(t_l) \sqsubset \mathrm{hso}_r$, for any output port $o_r$ in the simulation. Then, if input port $i_k$ is connected to output port $o_r$,

$$\mathrm{hsi}_k(t_l) = \mathrm{hso}_r(t_l) \sqsubset \mathrm{hso}_r = \mathrm{hsi}_k.$$

Furthermore, since any system input port will receive a packet with time greater than or equal to *tfinal*, and *tfinal* $> t_l$,

$$\mathrm{hsi}_k(t_l) \sqsubset \mathrm{hsi}_k,$$

for any system input port $i_k$. Combining these two facts,

$$\mathrm{hsi}_k(t_l) \sqsubset \mathrm{hsi}_k,$$

for *any* input port, $i_k$, in the system, whether it is connected to another module, or it is a system input port. No packets are produced in the simulation with time $t$ such that $t_l < t < t_{l+1}$, hence

$$\mathrm{hsi}_k(t_{l+1}) \sqsubseteq \mathrm{hsi}_k,$$

for any input port $i_k$ in the system. Therefore

$$\mathrm{Tfinal}(\mathrm{HSI}_j) \geq t_{l+1},$$

for any module $\mathbf{M}_j$. Lemma 1.2 can therefore be applied to show that

$$\mathrm{data}(\mathrm{HSO}_j(t_{l+1})) = \mathrm{HO}_j(t_{l+1}),$$

for any module $\mathbf{M}_j$.

b.) As has just been shown, if $t' = \mathrm{Tfinal}(\mathrm{HSI}_j)$ for the module $\mathbf{M}_j$, then $t' \geq t_{l+1}$. By the correct coordination requirement, for any output port $o_r$ of module $\mathbf{M}_j$, either

$$\mathit{tlast\text{-}out}_r > t' \geq t_{l+1},$$

or

$$tlast\text{-}out_r = \infty \geq t' \geq t_{l+1}.$$

That is, some packet with time value greater than $t_{l+1}$ will be produced on each output port, unless $t_{l+1} = \infty$. Thus, for any output port $o_r$ in the simulation, either

$$\mathbf{hso}_r(t_{l+1}) \sqsubset \mathbf{hso}_r,$$

or

$$t_{l+1} = \infty.$$

Therefore, by induction

$$\mathrm{data}(\mathrm{HSO}_j(tfinal)) = \mathrm{HO}_j(tfinal),$$

for any module $\mathrm{M}_j$ in the system.

Finally, to show that the module $\mathrm{M}_j$ would have the same final state $\mathrm{S}_f$ in both the simulation and the actual system, if $tfinal = \infty$, we have just shown that $\mathrm{data}(\mathrm{HSO}_k(tfinal)) = \mathrm{HO}(tfinal)$, for any module $\mathrm{M}_k$. Furthermore, for the system input ports, the statement of the theorem requires that $\mathrm{data}(\mathrm{SI}) = \mathrm{I}$. Thus, if the communication links between simulation modules operate correctly, and $tfinal = \infty$

$$\mathrm{data}(\mathrm{HSI}_j) = \mathrm{HI}_j,$$

for any module $\mathrm{M}_j$. By the statement of the theorem, $\mathrm{M}_j$ is started in the same initial state $\mathrm{S}_0$ in both the simulation and the actual system, therefore by the correct module simulation requirement, if $tfinal = \infty$ and the simulation module terminates, then both the simulation module and the actual module must have the same final state.

This completes the proof of the correctness of the simulation operations of Chapter 2 combined with the coordination operations of Chapter 3.

# Appendix 2

## Correctness of the Termination Operations

The following proof shows that the addition of the termination operations of Chapter 4 to the simulation modules will maintain the correctness of the simulation, with the added feature that the simulation will terminate once the termination conditions are satisfied.

---

Theorem 2. Correctness of Termination

a.) Suppose a simulation is performed in which the modules to be simulated obey the three requirements: functionality of output, monotonicity of output, and finite delay, and the simulation and coordination operations of each simulation module obey the three requirements: correct module simulation, correct ordering of output packets, and correct coordination, and furthermore the coordination operations of a simulation module cannot cause time packets $(\infty)$ to be sent out by the simulation module unless

$$\min_{1 \le k \le n} (tlast_k) = \infty.$$

Then the addition of termination operations to the simulation modules as described in Chapter 3 will not cause any of these requirements to be violated.

b.) If the actual system ever reaches a state in which no modules in the system will ever enter the firing mode unless more packets are received on the system input ports, then every simulation module in the simulation of this system will eventually produce time packets with value $\infty$ on all output ports, if all system input ports in the simulation receive time packets with value $\infty$.

---

## Proof of First Part

The termination operations will not affect the actual modules, hence the first three requirements for the Correctness of Simulation Theorem will hold. As for the correct module simulation requirement, the termination operations are designed not to interrupt the simulation of the modules. The only way they could potentially cause this requirement to be violated would be by terminating the simulation before the termination conditions are satisfied. Furthermore, since test packets contain no time values, their presence will not affect the correct ordering of output packets, or the correct coordination requirements. As long as the termination operations do not cause the simulation modules to send out time packets ($\infty$) before the termination conditions are satisfied, neither of these last two requirements will be violated either.

Since modules can communicate with each other only in the form of packets sent along the data channels, the conditions for termination for the modules in a connectivity class $C_j$ can be stated as:

1.) For each simulation module $M_i \in C_j$ all input ports $i_k$ such that $i_k \notin$ from_class$_i$ have received time packets ($\infty$).

2.) No simulation module $M_i \in C_j$ can simulate the firing of a module without receiving more data packets.

3.) No simulation module in $C_j$ will ever receive further data packets.

For a connectivity class which contains only one module and has no self-loop, there are no termination operations. Thus, as long as the termination operations for connectivity classes containing cycles do not cause the simulation modules in the class to terminate too soon, the correctness of the simulation will be maintained.

Termination operations might cause the simulation modules in a class to terminate prematurely in one of two ways. First, a test of the class might succeed, even though the termination conditions are not satisfied. Second, some simulation module $M_i$ might receive a time packet ($\infty$) on an input port $i_k \in$

from_class$_i$, before any test has succeeded, and then proceed to send out time packets ($\infty$) from all output ports, even though the termination conditions for the class are not satisfied. This second case can be ruled out rather easily. By the further restriction which has been placed on the coordination operations in the statement of the theorem, the coordination operations cannot cause a simulation module $M_i \in C_j$ to send out time packets ($\infty$) from its output ports, unless time packets ($\infty$) have been received on all input ports, including those in from_class$_i$. However, no simulation module $M_i \in C_j$ will receive a time packet ($\infty$) on an input port in from_class$_i$ unless some simulation module $M_l \in C_j$ sends a time packet ($\infty$) from an output port in to_class$_l$. Without any termination operations, this would happen only if $M_l$ had already received a time packet ($\infty$) on all input ports including those in from_class$_l$. Thus, no simulation module can be the first simulation module in the class to send time packets ($\infty$). Therefore the coordination operations alone cannot cause any simulation modules in a class to terminate if the class contains cycles. Furthermore, the termination operations cannot cause any simulation module in a class to send out time packets ($\infty$) until after a test has succeeded.

Thus, the proof of the first part of the theorem reduces to:

<u>Lemma</u> <u>2.1</u>. No Premature Termination

Suppose the termination control module T for a connectivity class $C_j$ has received time packets ($\infty$) on all input ports $i_k \in$ from_class$_T$, and no firing of the module can be simulated unless more data packets are received. If T sends out test packets (test.+) from all output ports $o_k \in$ to_class$_T$; receives K packets with value, test.+, in return, where

$$K = 1 + \sum_{M_i \in C_j} (|\text{to\_class}_i| - 1);$$

and it receives no further data packets while waiting for the returning test packets, this means that

> 1.) All simulation modules $M_i \in C_j$ have received time packets $(\infty)$ on all input ports $i_k \in$ from_class$_i$.
>
> 2.) No simulation module $M_i \in C_j$ can simulate the firing of a module without receiving more data packets.
>
> 3.) No simulation module in $C_j$ will ever receive further data packets.

---

The following sequence of assertions proves Lemma 2.1:

1.) If every simulation module $M_i \in C_j$ is *terminatable*, meaning that it receives a time packet $(\infty)$ on every input port which is not in from_class$_i$, and it eventually stops simulating the firing of the module, then during a test (or reset) of the class $C_j$

> a.) Each simulation module $M_i$ in $C_j$ will receive at least one test (or reset) packet.
>
> b.) Exactly K test (or reset) packets will be created, where
> $$K = 1 + \sum_{M_i \in C_j} (|\text{to\_class}_i| - 1).$$
>
> c.) At least one test (or reset) packet will be received on each input port in from_class$_i$ for every $M_i \in C_j$.

Assertion 1a) can be shown by induction on the length of the shortest path from T to $M_i$ (there must be a path from T to any other module in a connectivity class.) As a basis, if $l = 1$, then $T \rightarrow M_i$. $M_i$ will receive a test (or reset) packet shortly after T sends out test (or reset) packets from each output port $o_k \in$ to_class$_T$. Now assume the assertion is true for all simulation modules in the class with a path from T of length less than or equal to $l$. Then if there is a path of length $l$+1 from T to a simulation module $M_i$, there must be some module $M_p \epsilon C_j$, such that $M_p \rightarrow M_i$, and there is a path of length $l$ from T to $M_p$. Hence the induction assumption applies to $M_p$, meaning that it will receive at least one test packet. As long as $M_p$ is terminatable, it will send test (or reset) packets on every output port $o_k \epsilon$ to_class$_p$. Therefore, $M_i$ will eventually receive a test (or reset) packet.

Assertion 1b) follows directly from 1a). Initially, T creates and sends out |to_class$_T$| test (or reset) packets. The first time some other simulation module $M_i \epsilon C_j$ receives a test (or reset) packet, it will send out |to_class$_i$| test (or reset) packets, thereby creating |to_class$_i$| - 1 new ones. On receiving any further test (or reset) packet, a simulation module will send one test (or reset) packet, hence no new test packets will be created, nor will any be destroyed. By assertion 1a), eventually all simulation modules will receive at least one test (or reset) packet, therefore exactly K test (or reset) packets will be created, where

$$K = 1 + \sum_{M_i \epsilon C_j} (|\text{to\_class}_i| - 1).$$

Assertion 1c) also follows from 1a). Every input port $i_k$ in from_class$_i$ of a simulation module $M_i \epsilon C_j$ is connected to an output port $o_r$ of some module $M_l \epsilon C_j$, and $o_r$ is in to_class$_l$. By assertion 1a), $M_l$ will receive at least one test (or reset) packet. If $M_l$ is terminatable, it will eventually send a test (or reset) packet on every output port in the set from_class$_l$. Therefore, $M_i$ will eventually receive a test (or reset) packet on $i_k$. This is true for any input port $i_k$ in from_class$_i$ of any simulation module $M_i \in C_j$.

2.) If some simulation module $M_i$ is not terminatable, then less than K test packets will be created during a test, and therefore the test cannot succeed.

If $M_j$ is not terminatable, then it will not send out any test packets even if it receives any. Thus it will not create |to_class$_i$| - 1 test packets, which means that fewer than K test packets will be created during a test of the class. The test cannot succeed unless T receives K test packets, hence the test cannot succeed if some simulation module $M_j$ does not receive time packets ($\infty$) on all input ports which are not in from_class$_i$, or it does not stop simulating the firing of the module.

3.) For a test to succeed, no simulation module can receive any data packets between the time it receives its first test packet and the time it sends its last test packet.

If a simulation module did receive a data packet during this time, it would send out at least one packet (test.-). Once a (test.-) packet has been sent, the test must fail, because any terminatable simulation module which receives a (test.-) must send out a (test.-) packet. If all modules are terminatable, T will receive at least one (test.-) packet, and the test will fail. If some simulation module is not terminatable, the test will fail in any case.

4.) If a test succeeds, no simulation module $M_i \in C_j$ will receive any data packets after it has received its last test packet.

This will be shown by contradiction. Suppose a test of a class succeeds, but one or more simulation modules receive data packets after receiving their final test packets. Let $M_i$ be one of the first simulation modules for which this happens. That is, during the test, $M_i$ received all of its test packets and later receives a data packet p on some input port $i_k$, but this had not happened to any simulation module in the class before this point. If $i_k$ is not in $from\_class_i$, then $M_i$ could not have sent any test packets before receiving this data packet, because it cannot send any test packets before receiving a time packet ($\infty$) on $i_k$. Thus if a data packet is received on an input port $i_k$ which is not in $from\_class_i$ after any test packet has been received by $M_i$, either the simulation module would not be terminatable, or $M_i$ would send out a packet (test.-). In either case, the test would fail. Thus, $i_k$ must be in $from\_class_i$, which, by assertion 1c), implies that a test packet was received on input port $i_k$ before data packet p was received. By the first-in, first-out property of the communication links between simulation modules, some module $M_l$ must have sent data packet p to $M_i$ after it had sent a test packet to $M_i$. This possibility can be eliminated by looking at two cases:

Case 1. $M_l = T$

The termination control module T did not send out any test packets unless it could not simulate any more firings without receiving more data packets. Thus, in order for T to send data packet p after sending test packets, it must receive at least one data packet p' after the test has been initiated. Suppose data packet p' was received before the test has been completed. Then the test must fail by the rules for T. On the other hand, suppose packet p' was received after the test has been completed. Then T must have received all of its test packets and later received data packet p', before $M_i$ received data packet p from T. This violates the assumption that the receipt of p by $M_i$ was the first case in which a simulation module in the class received a data packet after receiving all of its test packets.

Case 2. $M_l \neq T$

In order for $M_l$ to send a test packet followed by data packet p to $M_j$, it must first receive a test packet, wait until no more firings can be simulated, and send the test packet to $M_i$. Then it must receive a new data packet p', simulate the firing of the module, and send data packet p to $M_i$. Thus, $M_l$ must have received data packet p' after it received its first test packet. Either this data packet was received before all test packets had been received by $M_l$, or it was received after this time. In the first case, $M_l$ would later receive a test packet and therefore send out a packet (test.-). By assertion 3) the test would fail in this case. In the second case, $M_l$ must have received p' on some input port after it had received all test packets, and this must have happened before $M_i$ received data packet p from $M_l$. This would violate the assumption that the receipt of p by $M_i$ was the first case in which a simulation module in the class received a data packet after receiving all of its test packets.

Thus, during a successful test, there is no simulation module $M_i$ which can be the first to receive a data packet after it has received all test packets.

5.) If a test succeeds, then no simulation module in the class can ever simulate a firing without receiving more data packets, nor will it ever receive more data packets.

If a test succeeds, then at the time a simulation module sent its first test packet, it could not simulate any more firings without receiving more data packets. By assertion 3), the simulation module did not receive any data packets between this time and the time at which it received its last test packet. By assertion 4), the simulation module did not, nor will it receive any data packets after the last test packet was received. Therefore, the test will succeed only if all simulation modules in the class are ready to be terminated.

This completes the proof that the addition of termination operations to the simulation modules cannot cause them to terminate too soon. Hence, none of the six requirements for the Correctness of Simulation Theorem of Appendix 1 can be violated. The correctness of the simulation will be maintained.

## Proof of the Second Part

Proving the second part of the theorem requires showing that the termination operations for each connectivity class will cause the simulation modules in the class to terminate, once the termination conditions for the class are satisfied. If a class $C_j$ consists of a single module $M_j$ which has no self-loop, then the correct coordination requirement will guarantee that time packets $(\infty)$ will be sent out once time packets with value $\infty$ have been received on all input ports, and no more firings of the module can be simulated.

Thus, this class will terminate once the termination conditions are satisfied. For connectivity classes containing cycles, it must be shown that once the connectivity class reaches the conditions for termination, any previous test or reset will be completed, a new test of the class will be initiated, and this test will succeed. These requirements are stated in the following lemma:

---

Lemma 2.2. Eventual Termination

A.) Completion of a Test or Reset

Suppose the termination control module T for a class $C_j$ sends a test (or reset) packet from each output port $o_k$ in to_class$_T$. If every simulation module $M_i$ in $C_j$ is *terminatable*, meaning it eventually receives time packets ($\infty$) on every input port $i_k$ which is not in from_class$_i$, and it eventually stops simulating the firing of the module, then all simulation modules in the class will receive at least one test (or reset) packet, and T will eventually receive K test (or reset) packets, where

$$K = 1 + \sum_{M_i \in C_j} (|\text{to\_class}_i| - 1).$$

B.) Eventual Success of Test

Suppose every simulation module $M_i$ in $C_j$ reaches a state in which time packets ($\infty$) have been received on all input ports which are not in from_class$_i$, no firings can be simulated without receiving more data packets, and no more data packets will ever be received by $M_i$. Then T will send out test packets (test.+) from all output ports in to_class$_T$, and it will eventually receive K (test.+) packets in return without receiving any further data packets.

C.) Termination after Successful Test

If T sends out time packets ($\infty$) on all of its output ports, then every simulation module $M_i$ in the class will eventually receive time packets ($\infty$) on all input ports and hence will terminate.

---

The following sequence of assertions proves each part of Lemma 2.2:

A.) Completion of a Test or Reset.

1.) If every simulation module in the class $C_j$ is terminatable, then

> a.) Each simulation module $M_i$ will receive at least one test (or reset) packet.
>
> b.) Exactly $K$ test (or reset) packets will be created.

These assertions are identical to assertions 1a) and 1b) in the proof of Lemma 2.1.

2.) If every simulation module in the class $C_j$ is terminatable, T will receive $K$ test (or reset) packets.

This follows from the way in which the signal output ports were chosen. Every simulation module except for T has a single signal output port. T has no signal output port. These ports are chosen in such a way that if we look only at the simulation modules in the class and the channels connected to their output ports, there is a path from every simulation module to T. Thus, the simulation modules and the channels connected to the signal output ports fulfill the necessary requirements for a directed tree [1], with each arc pointing from a son to its father. That is

> 1. There is a unique root node (namely T) with no arcs leaving from it;
>
> 2. Every other node ($M_i \neq T$) has a single arc leaving from it (namely the channel connected to the signal output port); and
>
> 3. There is a path from every node to the root node.

One important property of trees is that they are acyclic, hence there is no path, $M_i \xrightarrow{+} M_i$, which follows only signal output links. During the test (or reset) operations, K test (or reset) packets will be created, and once all simulation modules have received at least one test (or reset) packet, all test (or reset) packets will sent only from signal output ports. These packets will not be destroyed, nor can any terminatable simulation module hold onto them indefinitely, hence the packets can only be propogated toward the root node T. Therefore T will eventually receive all K test (or reset) packets, and the test (or reset) operations will be completed.

B.) Eventual Success of Test.

Suppose every simulation module $M_i$ in a class $C_j$ reaches a state in which time packets ($\infty$) have been received on all input ports which are not in from_class$_i$, no firings can be simulated without receiving more data packets, and no more data packets will ever be received by $M_i$.

1.) A new test of the class will be initiated.

If the simulation modules reach the above-mentioned state, they are all terminatable. Hence, by part A) of the lemma, any previous test or reset operations will be completed. Furthermore, during the reset operations every simulation module will receive a reset packet. Hence, any new test will take place as if no previous tests had occurred. Furthermore, once the reset operations are completed, a new test will be initiated.

2.) The test will succeed.

As long as no simulation module receives a data packet between the time it receives its first test packet and the time it receives its last test packet, it will send out (test.+) packets as long as it receives (test.+) packets. By our assumption, no simulation modules will receive data packets once the test has started. Therefore, since T starts the test by sending (test.+) packets, by part A) of the lemma, K (test.+) will be created, and T will eventually receive K (test.+) packets. Thus, the test will succeed once the termination conditions for the class are satisfied.

C.) <u>Termination after a Successful Test</u>.

Suppose the test of a class succeeds and T sends time packets ($\infty$) from all output ports.

1.) Every simulation module $M_i$ in $C_j$ will receive at least one time packet ($\infty$) on some input port $i_k$ in from_class$_i$.

This can be shown by induction on the length of the shortest path from T to $M_i$. In fact, the proof is virtually identical to the proof of assertion 1a) in the proof of Lemma 2.1.

2.) Every simulation module $M_i \in C_j$ will receive time packets ($\infty$) on every input port.

In order for the test to succeed, $M_i$ must have received time packets ($\infty$) on every input port which is not in from_class$_i$. Furthermore, by assertion 1) any module $M_l \in C_j$ connected to $M_i$ must receive at least one time packet ($\infty$) on some input port $i_r \in$ from_class$_l$. Hence, it will send out time packets ($\infty$)

on all output ports, including one to input port $i_k$ of module $M_l$. Therefore, all simulation modules in $C_j$ will receive time packets ($\infty$) on all input ports once the test has succeeded.

This completes the proof that the addition of the termination operations to the simulation modules will cause the simulation to terminate, once the termination conditions for the system are satisfied.