Proceedings of the 1988 Winter Simulation Conference
M. Abrams, P. Haigh, and J. Comfort (eds.)

A unified modeling methodology for performance evaluation of distributed discrete event simulation mechanisms

Bruno R. Preiss
Wayne M. Loucks

Department of Electrical Engineering
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1

V. Carl Hamacher

Departments of Electrical Engineering and Computer Science
University of Toronto
Toronto, Ontario, Canada, M5S 1A4

ABSTRACT

The main problem associated with comparing distributed discrete event simulation mechanisms is the need to base the comparisons on some common problem specification. This paper presents a specification strategy and language which allows the same simulation problem specification to be used for both distributed discrete event simulation mechanisms as well as the traditional single event list mechanism. This paper includes: a description of the Yaddes specification language; a description of the four simulation mechanisms currently supported; the results for three simulation examples; and an estimate of the performance of a communication structure needed to support the various simulation mechanisms. Currently this work has only been done on a uniprocessor emulating a multiprocessor. This has limited some of our results but lays a significant basis for future simulation mechanism comparison.

1. INTRODUCTION AND MOTIVATION

In this paper we advocate the use of a unified modeling methodology for the specification of discrete event simulations. The modeling methodology we have chosen is based on Chandy-Misra distributed discrete event simulation (Misra 1986). The principal benefit of our modeling methodology is that it is independent of the underlying simulation mechanism. In particular, it is possible to change the underlying simulation execution mechanism without altering the models. This allows us to compare the performance of various discrete event simulation mechanisms directly and quantitatively.

We have developed a simulation environment that supports four execution mechanisms:

1. traditional (event-list driven) discrete event simulation,
2. a distributed simulation mechanism using multiple, synchronized event lists,
3. Chandy-Misra distributed discrete event simulation (Misra 1986, Chandy and Misra 1987), and
4. Virtual Time based distributed discrete event simulation using the time warp mechanism (Jefferson 1985, Jefferson et al. 1987).

1.1. Deadlock Avoidance

A difficult problem in distributed discrete event simulation is the avoidance of deadlock (Chandy and Misra 1987). Various execution mechanisms have been proposed to avoid and/or detect deadlock, including null messages (Misra 1986), circulating marker (Misra 1986), and time warp (Jefferson et al. 1987). By providing a unified modeling methodology that allows the execution of exactly the same simulation under several different execution mechanisms, it is possible to directly and quantitatively compare the performance of various methods proposed for deadlock avoidance.

We also present an approach to deadlock avoidance in Chandy-Misra distributed simulation that relies on the programmer to explicitly control the behavior of the underlying execution mechanism based on a higher level understanding of the system being simulated. We show below that it is possible in some cases to avoid deadlock without increasing the number of messages that need to be sent.

2. MODELING METHODOLOGY

In this section we describe a modeling methodology that permits the use of different execution mechanisms. This methodology is based on the Chandy-Misra distributed discrete event simulation in that the real world system is decomposed into a network of interacting physical processes that are modeled by general state machines (Misra 1986). The principal advantage of this methodology is that it is independent of the underlying execution mechanism and the same simulation can be performed using various different execution mechanisms without modification.

The real-world system to be simulated is modeled as a static network of physical processes that periodically exchange information at discrete points in time. Each such exchange of information is called an *event*. Such networks are simulated by a collection of logical processes that exchange messages. Each message carries the information associated with an event and the time at which the event is to have occurred.

Logical processes are simply general state machines. The state of a logical process may change in response to the occurrence of an event. A logical process may create other events in response to an event. The description of the state and the behavior of a logical process is called a *model*. Thus, logical processes are instantiations of models.

A model is simply a combined state-transition and output event specification. A model specifies the next state and the output events that occur in response to a given input event or event combination. (An event combination is a collection of input events having the same time stamp.)

2.1. The Yaddes Language

We have devised a simulation specification language called Yaddes. Yaddes is a language in the style of Yacc (Johnson 1979) and Lex (Lesk 1979). I.e., the user constructs a specification of a system in the Yaddes language. A Yaddes parser translates this specification into a collection C language (Kernighan and Ritchie 1978) subroutines that can be compiled and linked with various execution mechanism libraries to construct a simulation program. A separate execution mechanism library is provided for each of the

mechanisms discussed above in Section 1 and described in detail in Section 3. Thus, a different execution mechanism can be used merely by linking the various object modules to a different library. The main components of a Yaddes program are *model specifications*, *process specifications*, and *connection specifications*. (A BNF specification of the Yaddes syntax is given in the appendix.)

Model Specifications: A model specification describes a class of physical processes. This description includes a specification of the number and names of the inputs and outputs of the physical process, a state type specification, an initial state value specification, and a list of action specifications. Each action specification consists of a list of input event combinations and a C language statement list. During the simulation, whenever a particular event combination occurs, the associated C language statement list is executed. Typically, the effect of the statements is to change the state and to generate future output events.

The Yaddes translator emits a C language subroutine and a C language type declaration for every model. When the subroutine is invoked by the execution environment and it is provided with a state variable of the specified type, the current value of simulation time, an array of input event values, and an event mask that specifies the input event combination. This subroutine causes the appropriate action statement list to be executed. As a consequence of executing an action statement list, the state variable may be assigned a new value and new events may be generated.

Process Specifications: A process specification describes a logical process. Logical processes are instantiations of models. Each instance of a model has a unique state with the type given in the model specification.

The Yaddes translator emits a C language variable declaration the type of which is determined by the model state specification. The state variable is initialized to the value given in the model specification.

Connection Specifications: Connection specifications are used to establish connections between the outputs and inputs of logical processes. Note that all connections are static. Each input to a process must be connected to exactly one output (unity fan-in). Each output from a process may be connected to several inputs (unlimited fan-out) or to none at all.

The Yaddes translator constructs a connection table for every output of each logical process. This table lists the inputs of logical processes to which the output is connected.

2.2. Example: Logic Simulation — Exclusive-OR Circuit

An example to illustrate the syntax of a Yaddes specification is shown in Figure 1. The system in this example is a logic network consisting of four NAND gates that implement the exclusive-OR function. The model called `TwoInputNand` describes the behavior of a NAND gate. `TwoInputNand` has two inputs called `in0` and `in1` and a single output called `out`. `TwoInputNand` has two state variables called `input0` and `input1`. These variables record the logic level (0 or 1) on the corresponding inputs.

The `TwoInputNand` model has an `initial` action and three event combination actions. The event combinations are (i) an event occurred on input `in0`, (ii) an event occurred on input `in1`, and (iii) simultaneous events occurred on both `in0` and `in1`.

```
#define GATE_DELAY 10
%%
model TwoInputNand
    inputs in0, in1
    outputs out
    state
    {
        int input0;
        int input1;
    }
    initial state { 0, 0 }
    action initial
    {
        OUTPUT ($out, $time + GATE_DELAY,
            NAND ($state->input0,
            $state->input1));
    }
    action in0
    {
        $state->input0 = $event [$in0];
        OUTPUT ($out, $time + GATE_DELAY,
            NAND ($state->input0,
            $state->input1));
    }
    action in1
    {
        $state->input1 = $event [$in1];
        OUTPUT ($out, $time + GATE_DELAY,
            NAND ($state->input0,
            $state->input1));
    }
    action in0, in1
    {
        $state->input0 = $event [$in0];
        $state->input1 = $event [$in1];
        OUTPUT ($out, $time + GATE_DELAY,
            NAND ($state->input0,
            $state->input1));
    }
end model

process X : ReadFromFile
process Y : ReadFromFile
process Gate0 : TwoInputNand
process Gate1 : TwoInputNand
process Gate2 : TwoInputNand
process Gate3 : TwoInputNand
process Z : WriteToFile

connect X.out to Gate0.in0, Gate1.in0
connect Y.out to Gate0.in1, Gate2.in1
connect Gate0.out to Gate1.in1, Gate2.in0
connect Gate1.out to Gate3.in0
connect Gate2.out to Gate3.in1
connect Gate3.out to Z.in
```

Figure 1: Yaddes Specification Example — Exclusive-OR Circuit

A sequence of C language statements is associated with each event combination. Certain metavariables are available for use within the C language statement lists. Metavariables begin with the symbol $. Associated with each input and output of the model is a metavariable with the same name.

The value of the metavariable is the number of the input or output. Other metavariables include `$time` which contains the current simulation time and `$event` which is an array containing the values of the input events.

Certain functions are also available for use within the C language statement lists. In this example, the function OUTPUT is used to cause events on the output of the model. The arguments are (i) an output number, (ii) a (future) time, and (ii) an event value. This function uses the connection tables constructed by the Yaddes translator to send output events to the appropriate logical process inputs.

A total of seven logical processes are declared in Figure 1. In addition to the four NAND gates, there are two instances of the `ReadFromFile` model and an instance of the `WriteToFile` model. (The specification of these models has been omitted.)

2.3. Special Functions

In this section we describe a number of functions that can be invoked in action statement lists to control the execution of the simulation. The first of these, the OUTPUT function was introduced above. (The NULLOUTPUT function has a similar purpose.) The other functions, IGNORE, DEACTIVATE, and ACTIVATE, are associated with the prevention of deadlock in the Chandy-Misra execution mechanism. These functions do not alter the meaning of the simulation specification. I.e., they do not create or destroy events and they do not alter the states of processes.

OUTPUT and NULLOUTPUT Functions: The OUTPUT and NULLOUTPUT functions are used to emit an event on an output of a logical process. These functions require three arguments: (i) an output number, (ii) a (future) simulation time, and (ii) an event value. These functions place an event with the specified parameters on the inputs of the logical processes to which the specified output is connected. (The connection information is obtained from the connection table for the specified output.)

It should be noted that the operation of the OUTPUT and NULLOUTPUT routines is a function of the simulation environment. For example in the event list environment an OUTPUT simply inserts an event into the future events list while in the Chandy-Misra environment a message must be transferred between processors.

The OUTPUT and NULLOUTPUT functions are functionally identical in the Chandy-Misra execution environment. In the other execution environments described below, the NULLOUTPUT function has no effect. Thus, by using the NULLOUTPUT function, it is possible to identify the superfluous events required by the Chandy-Misra execution environment to avoid deadlock.

IGNORE Function: The IGNORE function is used to dynamically disable an input to a logical process. This function requires two arguments: (i) an input number and (ii) a (future) simulation time. The effect of this function is to inform the execution environment that no input event will occur on the specified input until after the given simulation time. This function is typically used in the prevention of deadlock in the Chandy-Misra distributed discrete event execution environment (discussed below). It has no effect in the other execution environments.

DEACTIVATE Function: The DEACTIVATE function is used to dynamically disable an input to a logical process. This function requires one argument: an input number. The effect of this function is to inform the execution environment that, until further notice, no input event will occur on the specified input. This function is typically used in the prevention of deadlock in the Chandy-Misra distributed discrete event execution environment (discussed below). It has no effect in the other execution environments.

ACTIVATE Function: The ACTIVATE function is used to dynamically enable an input to a logical process previously disabled using the DEACTIVATE function. This function is typically used in the prevention of deadlock in the Chandy-Misra distributed discrete event execution environment (discussed below).

Note that it is the programmer's responsibility to guarantee that the system being simulated is not affected by the use of the IGNORE, DEACTIVATE, and ACTIVATE routines. In particular, it is an error for an event to occur on an IGNOREd or DEACTIVATEd input. The execution environment warns the user if a message is inadvertently being ignored but will make no attempt to recover.

3. SIMULATION ENVIRONMENTS

In this section we describe the four execution environments currently supported. As stated above, these execution environments exist in the form of libraries to which compiled Yaddes object modules are linked. Since exactly the same simulation can be executed under different execution mechanisms, a direct and quantitative comparison of the performance of the various mechanisms can be made.

3.1. Event List Driven Simulation

The event list driven simulation environment uses the traditional discrete event simulation mechanism. A single data structure, called the *event list*, is used to hold future events. Future events are sorted by time. The basic execution cycle involves removing events from the event list, forming event combinations, and causing the appropriate logical processes to perform the action associated with the given input event combination. When an action invokes the OUTPUT function, it causes events to be inserted into the future event list.

3.2. Multiple, Synchronized Event Lists

The multiple, synchronized event list execution environment is a simple extension of the basic event list mechanism for execution on a multiprocessor. In this mechanism, each processor has its own future event list. In addition, one processor has special status and acts as a global scheduler. The basic execution cycle is somewhat more complex in order to guarantee correct execution on the multiprocessor.

First, each processor sends a message to the scheduler indicating the simulation time of the next event on its event list. The scheduler selects the minimum next event time and sends a message to all the processors containing this value. Each processor having this minimum value removes events from its event list, forms event combinations, and invokes the appropriate logical processes' actions. When an action invokes the OUTPUT function, it either causes an event to be inserted into the local future event list, or it sends a message to a remote processor requesting that it insert an event into its future event list. When a processor is finished executing all the actions for a given value of simulation time, it sends a completion message to all its successors indicating that it is

done. Finally, the processor waits until it receives a completion message from all its predecessors. (A processor deduces which processors are its successors and predecessors from the output tables). At this point the execution cycle is complete and may begin again.

In this mechanism, each logical process is statically assigned to a processor. This assignment is specified in the Yaddes source. In addition to the information described above, the output tables also contain information that allow the OUTPUT function to determine whether a local or remote event posting is required.

This execution environment currently runs under a single UNIX† process that simulates a multiprocessor environment by multitasking. Since the multitasking system is implemented in one process, a full UNIX context switch is not required to change tasks.

3.3. Chandy-Misra Distributed Discrete Event Simulation

In the Chandy-Misra distributed discrete event simulation execution environment each logical process runs as a separate task on a separate processor. In this environment, the logical processes are called *Envelopes*. A model instantiation is associated with each envelope. Envelopes exchange messages containing events.

The basic execution cycle begins when an envelope receives a message. The envelope buffers messages until an event combination can be formed. (An event combination with simulation time t can only be formed when an envelope has received an event message for each input of its associated model having time $t' \geq t$.) When an event combination is formed, the appropriate action is invoked. When an action invokes the OUTPUT function, it causes the envelope to send event messages to the envelopes specified in the output table. This execution environment has the potential for deadlock. We have not yet implemented the deadlock detection/recovery scheme discussed in (Misra 1986). We require the simulation programmer to explicitly avoid deadlock.

This execution environment currently runs under a single UNIX process that simulates a multiprocessor environment by multitasking.

3.4. Virtual Time based Distributed Discrete Event Simulation

The virtual time based distributed discrete event simulation mechanism is based on Jefferson's time warp operating system (Jefferson et al. 1987). As in the Chandy-Misra mechanism, each logical process runs as a separate task on a separate processor and is called an Envelope. A model instantiation is associated with each envelope and envelopes exchange messages containing events.

The basic execution cycle begins when an envelope receives a message. When a message arrives, there are two possibilities — its time stamp is either before or after the current (local) value of simulation time. If its time stamp is after the current time, an input event combination is formed and the appropriate action is invoked. If its time stamp is before the current time, the envelope backs up to the time on the incoming message. This backing-up is facilitated by an elaborate checkpointing mechanism described in (Jefferson 1985) and (Jefferson et al. 1987). Essentially, an earlier state is restored, input event combinations are rescheduled, and output events are cancelled by sending *antimessages*.

This execution environment currently runs under a single UNIX process that simulates a multiprocessor environment by multitasking.

† UNIX is a trademark of AT&T.

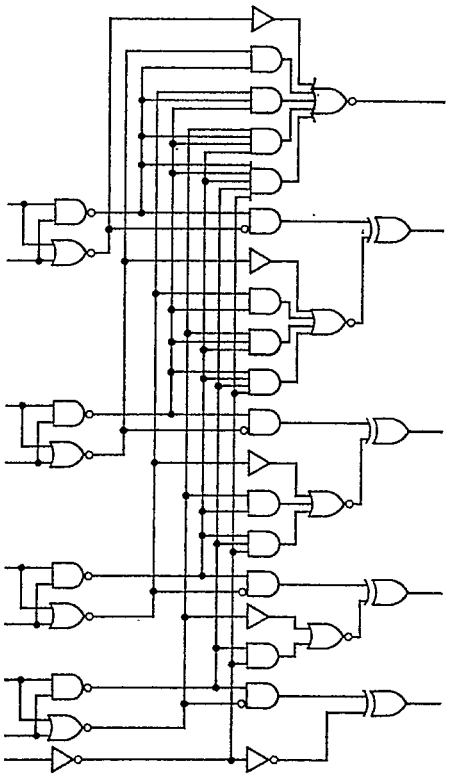

Figure 2: 4-Bit Binary Adder
(74LS283) (National Semiconductor 1981)

4. BENCHMARK SIMULATIONS

In order to evaluate the various execution environments described above, three different systems were simulated. These systems are: an acyclic logic circuit (a 4-bit adder); a cyclic logic circuit (a 4-bit counter); and a cyclic system (the P-Bus — a multiprocessor communication structure). A summary of their characteristics is given in Table 1 and system level diagrams in Figures 2-4. Based on these simulations, performance metrics were established to compare the potential performance of the execution mechanisms for other simulation problems. It has been our goal to specify reasonable sized, realistic problems which on the surface appear to have a significant amount of parallelism. In section 5 below the performance of these benchmarks is discussed.

4.1. Acyclic Logic Simulation — 4-bit Binary Adder

This is a simulation of the logic circuit of a 74LS283 4-bit binary adder. This circuit was chosen because it is acyclic, and therefore does not encounter deadlock problems when using the Chandy-Misra distributed discrete event simulation environment. The circuit was simulated for a total of 801 input events. The time between events was such that the outputs of the simulated chip stabilized between events.

Table 1: Benchmark Simulation Characteristics

Type	Acyclic Logic	Cyclic Logic	Cyclic System
Name	4-Bit Adder	4-Bit Counter	P-Bus
Number of `model` specifications	15	12	7
Number of `process` specifications	50	58	49
Number of `connect` specifications	45	57	85
Number of External Inputs	9	9	0
Execution Time (seconds)	6-119	3-91	77-1200

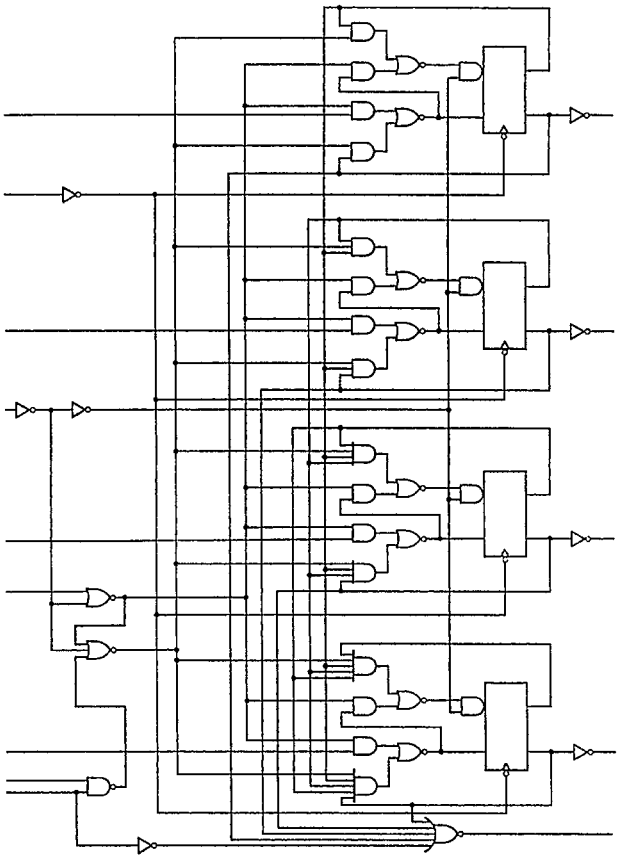

Figure 3: 4-bit Synchronous Counter (74LS63) (National Semiconductor 1981)

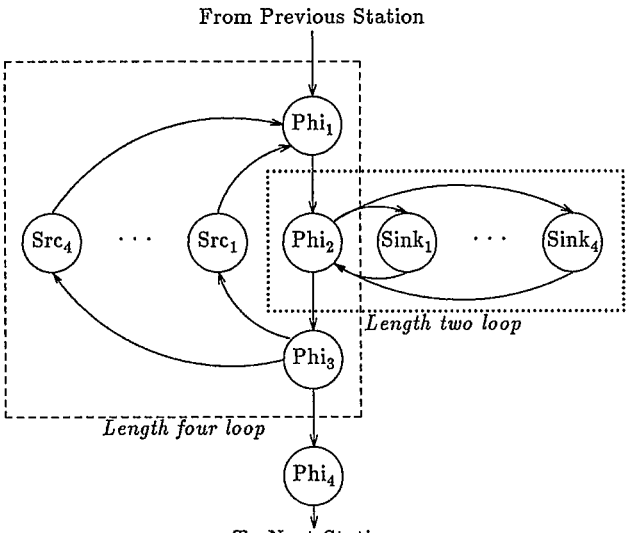

Figure 4: P-bus Station Model

4.2. Cyclic Logic Simulation — 4-bit Synchronous Counter

This is a simulation of the logic circuit of a 74LS163 4-bit synchronous counter. This circuit was chosen because it is cyclic, and therefore poses a deadlock problem when using the Chandy-Misra distributed discrete event simulation environment. Deadlock was avoided in this circuit solely through the use of null messages. The circuit was simulated for a total of 211 input events. The time between events was such that the outputs of the simulated chip stabilized between events.

4.3. System Simulation — P-Bus

The P-Bus is a multiprocessor interconnection scheme, originally proposed for use in a tightly coupled multiprocessor FERMTOR (Loucks and Vranesic 1980, Rose, Loucks, and Vranesic 1985). It is composed of a number of stations (four in this case) each of which has a number of processors (four in this case) attached to a bus segment. The bus segments are interconnected in a ring fashion. Each bus segment is controlled by a station manager. The operation of the manager is represented as the processes Phi_1 to Phi_4 which may be considered as the four phases of a global clock. The other processes, Src_i and $Sink_i$, represent exponential service time processes which respond to an input message with an output message with an exponentially distributed random delay. The delay is controlled by a pseudorandom number generator†.

In the implementation of this system several alternatives for deadlock control were investigated. These methods affect the two loops shown in Figure 4 and labeled as a *length-two loop* and a *length-four loop.*

In the Chandy-Misra environment, the length-two loop would deadlock immediately without the addition of some deadlock avoidance mechanism. The sink accepts the first message, finishes servicing it at a later time (sending a message to Phi_2 indicating completion) and then sends no further messages until the next message arrives for that sink. In the interim Phi_2 is blocked from processing the next

† Although not discussed above, the Yaddes environment provides a facility to permit each process to have one (or more) independent pseudorandom number streams. This was found to be necessary to produce reproducible simulation results under the various different execution mechanisms.

Table 2: Effects of Various Deadlock Avoidance Techniques

Method	NULL MESSAGES	IGNORE	ACT/DEACT	ACT/DEACT
Loops Affected	—	2	2	2 and 4
Total Number of Messages	377,849	241,312	241,312	115,433
Number of Null Messages	262,464	125,925	125,925	48
Simulated Time (clock cycles)	2,000	2,000	2,000	2,000
Execution Time (Seconds)	308	207	202	108

incoming message from Phi_1, as not all of its inputs have advanced to an appropriate time. To overcome this problem, three techniques that involve modification of the models were tried.

Null Messages: Null messages are sent on every cycle of Phi_2 to indicate to the sinks that "there is no message at this time," and in response to this each sink responds with "I will not acknowledge a message for a specified time."

Ignore Input Until time x: The null message scheme, although simple, creates a large number of messages and fails to exploit the programmer's knowledge of the operation of the P-bus. There will be no message from $Sink_i$ until there is a message delivered to $Sink_i$. Hence a sink can be IGNOREd if no message is sent to that sink. The effect of using IGNOREs is that eight null messages (one to and from each of four sinks) are replaced with four calls to IGNORE for each tick of the global clock.

Deactivate Input: Instead of using repeated IGNORE calls, it is possible to use pairs of DEACTIVATE and ACTIVATE calls to avoid deadlock. This scheme increases efficiency somewhat by reducing the number of repeated subroutine calls.

The latter two alternatives result in an increased state size for the Phi_2 model as it has to remember the current status of all of its sinks.

The ACTIVATE/DEACTIVATE scheme can also be used to eliminate the null messages in the length-four loop. In this case Phi_1 is required to predict the operation of the Phi_3 process in order to selectively deactivate the appropriate source input. This is possible as a result of a more detailed understanding of the system being modeled, since it is known that each source can have at most one outstanding message, and that once that message is transferred to Phi_1, the input from that source can be ignored until that message is acknowledged by Phi_3.

The effect of these various techniques is given in Table 2. Note that the use of ACTIVATE/DEACTIVATE pairs in both length-two and length-four loops results in the smallest number of messages. (This is the version discussed in Section 5.)

5. PERFORMANCE MEASUREMENTS

The benchmark systems described above were run using the 4 run-time environments described in section 2. All of these tests were done using a DEC μVAX II† computer system running BSD4.3 UNIX. The distributed execution environments were each run under one UNIX process, which maintained several apparent sub-processes. These subprocesses have a data stack maintained by the main program and pseudo-context switching is accomplished by small assembler routines.

The goal of this performance study thus far has been to examine the communication needs and the potential improvement needed to justify a tightly coupled distributed discrete event simulation system.

5.1. Performance Metrics

As a result of running on a uniprocessor only characteristics such as the number of messages can be accurately measured. It is necessary to introduce a time-slice regimen to imitate the operation of a multiprocessor. The time slice regimen clouds any attempt to accurately estimate the potential parallelism for a given system. What can be done quite effectively is to estimate the following.

1. The additional overhead needed to execute the simulation in the various environments. This has not been easy in the past because the problems had to be represented differently for the different schemes. (E.g., see (Reed, Malony, and McCredie 1988).)
2. The parallelism needed to provide any improvement over the sequential simulation.
3. The performance requirements of the communication structure.

The raw results from the three simulations are given in Tables 4-6. These results are examined as they relate to the problem simulated in the sections below and in section 5.5 the overall performance of the simulation environments is examined in more detail. Table 3 defines the various performance measures used in Tables 4-6.

The only reliable estimate of the potential for parallel performance is an estimate of the average number of items at the head of the event list with the same time. In the case of the adder and the counter this value is 4.8 indicating that there is slightly more than one operation per bit that can be done. In the case of the P-Bus this number is 4.7 indicating that on average there is slightly more than one activity in progress in each of the four stations. This indicates that the best performance numbers to compare are the single event list, the multiple event list with 4 separate event lists, the Chandy-Misra and the Virtual Time systems. It is also clear from the tables that one event list per process has an enormous overhead associated with synchronizing the event lists while fewer than four event lists fails to capture the average parallelism which is available in the problem.

In the performance tables it can be seen that all of the distributed schemes perform (on a uniprocessor) worse than the single event list mechanism. In the sections below we examine the parallelism necessary to speed up the distributed mechanisms to make them equal to the single event list. These estimates are very rough and *ignore* the added overhead needed to transfer messages to their destinations.

† DEC and VAX are trademarks of the Digital Equipment Corporation.

Table 3: Performance Measures

T_{model}	The time spent executing code directly related to the model. (This excludes scheduling and messaging time).
T_{ex}	The total execution time of the simulation.
T_{model}/T_{ex}	The fraction of time spent doing useful things. This assumes that all non-model time is overhead. This includes such times as the time to add new events to event lists and the time needed to send messages.
T_{ex}/T_{Single}	This is a measure of the parallelism that would be needed to match the speed of the single event list (i.e. the uniprocessor case). This ratio should be viewed as a lower bound on the necessary parallelism as these times ignore communication time, which is discussed in section 5.5.
N_{mess}	The number of non-overhead messages needed for the simulation. This represents the minimum number of messages needed to run the simulation
N_{oh}	The number of overhead messages need. This includes the Chandy Misra null messages, the multiple event list synchronization messages and the messages needed to maintain the Virtual Time system (antimessages, token messages, and messages cancelled by backing-up to an earlier state).
N_{MC}	The total number of model calls needed for the simulation.
$(N_{mess}+N_{oh})/N_{MC}$	The average number of messages sent per model call.

Table 4: Performance Results from the 4-Bit Adder

	Single Event List	Multiple Event Lists 1	2	4	50	Chandy-Misra	Virtual Time
T_{model}	2.02	1.96	2.16	2.22	1.86	2.58	2.86
T_{ex}	6.3	7.6	10.2	15.1	119.	16.0	30.9
T_{model}/T_{ex}	0.32	0.26	0.21	0.15	0.02	0.16	.09
T_{ex}/T_{Single}	1	1.2	1.6	2.4	19.	2.6	4.9
$N_{mess}/1000$	—	0	1.8	2.5	9.16	10.0	9.97
$N_{oh}/1000$	—	3.7	11.2	26.2	350.	8.4	2.29
$N_{MC}/1000$	9.0	9.0	9.0	9.0	9.0	13.2	11.2
$(N_{mess}+N_{oh})/N_{MC}$	—	.42	1.17	3.19	39.9	1.4	1.09

In section 5.5 the communication requirements of the various schemes are examined.

5.2. Binary Adder

In the adder both the Virtual Time and the Chandy-Misra systems require extra model calls when compared to the single event list. These calls result from the null messages needed to synchronize the Chandy-Misra system and the model calls which must be re-executed in the Virtual Time system.

In this case only the virtual time mechanism requires a parallelism greater than 4.8 to achieve any speed up over the uniprocessor case. Although there is only some relationship between the number of events with the same time on the single event list and the parallelism that can be obtained from the virtual time mechanism, this result indicates that to be at all effective (in this case) the virtual time system must discover a significant degree of parallelism not obvious in the problem. It is also worth noting that the multiple event list (with 4 lists) needs approximately the same time to execute as Chandy-Misra even though there is a significantly larger number of messages in the multiple event list structure.

5.3. Synchronous Counter

The main point of interest in the counter results is that the number of null messages needed to support the Chandy-Misra model swamps any potential for parallelism. This is a result of there being no attempt to use IGNORE or DEACTIVATE/ACTIVATE constructs to reduce the need for null messages. The gate models simply forward incoming null messages to the output after an appropriate delay. In this case, that of a very cyclic graph, both the Chandy-Misra and the Virtual Time scheme require a large degree of parallelism just to achieve the same speed as the single event list.

5.4. System Simulation — P-Bus

The P-Bus model illustrates the Chandy-Misra model of execution at its best. In this case the number of model calls is the same in both the single event list case and in Chandy-Misra. As a result there are fewer messages in the Chandy Misra case than in any of the other distributed schemes with which it is compared.

In the case of this moderately cyclic graph, the inherent parallelism of 4.7 is sufficient for the distributed schemes to achieve improved performance.

Table 5: Performance Results from the 4-Bit Counter

	Single Event List	Multiple Event Lists 1	2	4	58	Chandy-Misra	Virtual Time
T_{model}	0.78	0.87	0.95	0.81	0.90	6.89	1.18
T_{ex}	2.8	3.3	4.5	7.0	60.2	89.2	17.8
T_{model}/T_{ex}	0.28	0.26	0.21	0.12	0.02	0.08	.07
T_{ex}/T_{Single}	1	1.2	1.6	2.5	21.	31.	6.4
$N_{mess}/1000$	—	0	1.23	1.77	4.06	4.38	4.36
$N_{oh}/1000$	—	1.6	4.8	13.6	177.	119.	5.7
$N_{MC}/1000$	3.8	3.8	3.8	3.8	3.8	76.6	9.4
$(N_{mess}+N_{oh})/N_{MC}$	—	0.42	1.58	4.01	47.2	1.61	1.07

Table 6: Performance Results from the P-Bus

	Single Event List	Multiple Event Lists 1	2	4	48	Chandy-Misra	Virtual Time
T_{model}	18.5	17.8	18.0	18.0	18.0	18.7	28.9
T_{ex}	77.	85.	107.	146.	1264.	115.	277.
T_{model}/T_{ex}	0.24	0.21	0.17	0.12	0.01	0.16	.1
T_{ex}/T_{Single}	1	1.1	1.4	1.9	16.	1.5	3.6
$N_{mess}/1000$	—	0	10.0	20.0	115.3	115.4	115.4
$N_{oh}/1000$	—	40.	120.	280.	4,640.	0.05	25.
$N_{MC}/1000$	94.	94.	94.	94.	94.	94.	178.
$(N_{mess}+N_{oh})/N_{MC}$	—	.42	1.38	3.18	50.4	1.22	0.79

5.5. Comments on the Communication Requirements for the Various Simulation Mechanisms

We have not yet run the distributed simulations on a distributed system (see section 7) and as a result our parallelism measures can only be viewed as estimates. However, we can estimate the required performance of a communication structure to support this intensity of communication. If we assume that all of the communication can be done in parallel with the computation and that we are fortunate enough to partition the problem such that the computation and communication load is evenly distributed (truly an optimistic assumption) then we can calculate the needed transfer time of a message.

$$\text{Time}_{\text{for a single message tranfer}} = \frac{T_{ex}/\text{Parallelism}}{N_{mess} + N_{oh}}$$

Where the Parallelism is the average parallelism factor calculated from the single event list case (4.8 for the adder and the counter and 4.7 for the P-Bus). This calculation also assumes that the interprocessor communication structure is serial in nature (i.e., that there can be at most one message in transit at any given time).

The results of this measure indicate that the message passing system would have to pass messages between 95 μseconds (for the counter in the multiple event list environment) and 420 μseconds (for the P-Bus in the virtual time environment). Although these are only estimates they indicate that process level distributed simulation should only be attempted if very fast (process to process) messaging times are possible. It is worth noting that the P-Machine (Jager and Loucks 1987, Jager 1987) achieves a time of 50 μseconds for short messages.

6. CONCLUSIONS

We have demonstrated that it is possible to use a unified modeling methodology for distributed discrete event simulation. Although this modeling methodology is based on the Chandy-Misra distributed discrete event simulation mechanism, our simulation performance measurements indicate that this choice of modeling methodology does not favor the Chandy-Misra simulation mechanism.

7. FUTURE WORK

We are now in the process of porting the Yaddes parser and simulation mechanism kernels to a network of APOLLO† workstations connected by an APOLLO token ring. This will allow the direct measurement of the simulation performance as a function of the number of processors. Since our unified modeling methodology supports various different execution mechanisms, we will be able to directly compare them.

It is our intention to use the remote procedure call mechanisms available in Apollo's Network Computing System®† (Apollo 1987) to support the various distributed execution mechanisms. The simulation environment libraries will be written using NIDL (network interface definition language) (Apollo 1987) specifications for the remote procedure call interfaces.

ACKNOWLEDGEMENTS

This research was funded by the Information Technology Research Centre of the Province of Ontario (Canada) and the Natural Sciences and Engineering Research Council of Canada under grants A5192, A6840 and OGP0036635.

† APOLLO and Network Computing System are trademarks of Apollo Computer Inc.

AUTHORS' BIOGRAPHIES

Bruno R. Preiss received the B.A.Sc degree in engineering science in 1982, the M.A.Sc degree in electrical engineering in 1984, and the Ph.D. degree in electrical engineering from the University of Toronto, Toronto, Ont. Canada. He is an Assistant Professor in the Department of Electrical Engineering at the University of Waterloo, Waterloo, Ont., Canada. His current research interests include multiprocessor and parallel processor computer architectures, dataflow, and distributed simulation.

Wayne M. Loucks received the B.A.Sc. degree in electrical engineering in 1975 from the University of Waterloo, Waterloo, Ont., Canada, and the M.A.Sc degree in electrical engineering in 1977 and the Ph.D. degree in electrical engineering in 1980 from the University of Toronto, Toronto, Ont. Canada. He is an Associate Professor in the Department of Electrical Engineering at the University of Waterloo, Waterloo, Ont., Canada. His current research interests include tightly coupled multiprocessor structures to support fine granularity co-operation among processes and distributed simulation.

V. Carl Hamacher received the B.A.Sc degree in engineering physics in 1963 from the University of Waterloo, Waterloo, Ont., Canada, the M.Sc. degree in electrical engineering in 1965 from Queen's University, Kingston, Ont., Canada, and the Ph.D. degree in electrical engineering in 1968 from Syracuse University, Syracuse, NY. He is a Professor in the Departments of Electrical Engineering and Computer Science at the University of Toronto, Toronto, Ont., Canada. His current research interests include local area computer networks and real-time computer systems. He is a coauthor of the textbook *Computer Organization* (McGraw-Hill: New York, 1984).

APPENDIX A: YADDES BNF SPECIFICATION

file ::= *preamble specifications postamble*

preamble ::= ϵ | *program*

specifications ::= `%%` *modelList processList connectionList procedureList*

postamble ::= ϵ | `%%` *program*

modelList ::= ϵ | *modelList model*

processList ::= ϵ | *processList process*

connectionList ::= ϵ | *connectionList connection*

procedureList ::= ϵ | *procedureList procedure*

model ::= `model` *identifier inputPart outputPart statePart initialStatePart actionList* `end model` |
`external model` *identifier inputPart outputPart statePart initialStatePart* `end model`

process ::= `process` *identifier* : *identifier* |
`process` *identifier* `on` *constant* : *identifier*

connection ::= `connect` *port* `to` *portList*

procedure ::= `procedure` *identifier* : `initial` |
`procedure` *identifier* : `final`

inputPart ::= `inputs` *nameList* | `inputs none`

outputPart ::= `outputs` *nameList* | `outputs none`

statePart ::= `state {` *struct-decl-list* `}`

initialStatePart ::= `initial state {` *initializer-list* `}`

actionList ::= ϵ | *actionList action*

port ::= *identifier* . *identifier*

portList ::= *port* | *portList* , *port*

nameList ::= *identifier* | *nameList* , *identifier*

action ::= *actionHeaderList* `{` *statement-list* `}`

actionHeaderList ::= *actionHeader* |
actionHeaderList actionHeader

actionHeader ::= `action initial` | `action final` |
`action default` | `action none` |
`action` *nameList*

program ::= *see Kernighan & Ritchie, p. 218.*

identifier ::= *see Kernighan & Ritchie, p. 179.*

constant ::= *see Kernighan & Ritchie, p. 180.*

struct-decl-list ::= *see Kernighan & Ritchie, p. 216.*

initializer-list ::= *see Kernighan & Ritchie, p. 217.*

statement-list ::= *see Kernighan & Ritchie, p. 217.*

REFERENCES

Apollo (1987). "Network Computing System (NCS) Reference," Order No. 010200, Apollo Computer Inc., Chelmsford, MA.

Chandy, K. M. and Misra, J. (1987). "Conditional Knowledge as a Basis for Distributed Simulation," Report No. 5251:TR:87, Computer Science Department, California Institute of Technology, Pasadena, CA.

Jager, W.J. and Loucks, W.M. (1987). "The P-MACHine: A Hardware Message Accelerator for a Multiprocessor System," *Proceedings of the 1987 International Conference on Parallel Processing*, Pennsylvania State University Press, pp. 600-609.

Jager, W.J. (1987). "The P-MACHine: A Hardware Message Accelerator for a Multiprocessor System," M.A.Sc. Thesis, CCNG Technical Report T-165, University of Waterloo, Department of Electrical Engineering, Waterloo, Ontario.

Jefferson, D., Beckman, B., Wieland, F., Blume, L., DiLoreto, M., Hontalas, P., Laroche, P., Sturdevant, K., Tupman, J., Warren, V., Wedel, J., Younger, H., and Bellenot, S. (1987). "Distributed Simulation and the Time Warp Operating System," *Proceedings of the 12th SIGOPS — Symposium on Operating Systems Principles*, pp. 77-93.

Jefferson, D. R. (1985). "Virtual Time," *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, pp. 404-425.

Johnson, S. C. (1979). "Yacc — Yet Another Compiler Compiler," in *Unix Programmer's Manual*, Bell Laboratories, Murray Hill, NJ.

Kernighan, B. W. and Ritchie, D. M. (1978). *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ.

Lesk, M. E. (1979). "Lex — A Lexical Analyzer Generator," in *Unix Programmer's Manual*, Bell Laboratories, Murray Hill, NJ.

Loucks, W. M. and Vranesic, Z. G. (1980). "FERMTOR; A Flexible Extendible Range Multiprocessor," *CIPS Session 80*, Canadian Information Processing Society, Victoria, B.C., pp. 134-149.

Misra, J. (1986). "Distributed Discrete-Event Simulation," *ACM Computing Surveys*, Vol. 18, No. 1, pp. 39-65.

National Semiconductor (1981). *Logic Databook,* National Semiconductor Corp., Santa Clara, CA.

Reed, D. A., Malony, A. D., and McCredie, B. D. (1988). "Parallel Discrete Event Simulation Using Shared Memory," *IEEE Transactions on Software Engineering*, Vol. 14, No. 4, pp. 541-553.

Rose, J., Loucks, W., and Vranesic, Z. (1985). "FERMTOR: A Tuneable Multiprocessor Architecture," *IEEE Micro*, Vol. 5, No. 4, pp. 5-17.