Proceedings of the 1985 Winter Simulation Conference
D. Gantz, G. Blais, S. Solomon (eds.)

THE IMPLEMENTATION OF THE HIERARCHICAL ABSTRACT SIMULATOR ON THE HEP COMPUTER

Arturo I Concepcion
Department of Computer Science
Michigan State University
E. Lansing, MI 48824

A methodology is being developed to map the hierarchical abstract simulator onto distributed simulator architectures. The hierarchical abstract simulator is a multicomponent, multilevel discrete event models communicating via message passing. This paper reports on an alternative mapping realization of the hierarchical abstract simulator by using DENELCOR's FORTRAN 77, an extension of FORTRAN 77 for parallel programming, on the Heterogeneous Element Processor (HEP) computer. Several runs were made on the implementation and it was found out that there are three constraints that affect the performance (execution time) of the HEP implementation: number of processors available, degree of synchronization and intercommunication, and workload.

1. Introduction

A distributed simulation methodology based on Discrete Event Specification System, DEVS, [9] was introduced in Concepcion [2] in which multicomponent discrete event models may be simulated by employing multiprocessor architectures. The main thrust of that research is the mapping of the hierarchical multicomponent models onto distributed simulators so that correct and efficient simulation is obtained. The advantages of such distributed simulators over conventional sequential simulation are:

1. The mapping of a network of discrete event components onto the network of processors can better preserve its structure. In the best case, each processor might represent a single model component. This enhances comprehension of the simulator-model relationship, and therefore also, simulation experimentation and model exploration.
2. Advantage may be taken of intrinsic parallelism in the operation of model components by having concurrent execution by each processor of its component's state transitions.

The distributed simulation methodology consists of 5 layers and 4 steps. The lowest layer is the real system to be simulated. By means of the DEVS formalism, the real system is specified as a distributed model. This produces the second layer. From the specification of the distributed model, a transformation is applied to obtain the hierarchical abstract simulator. This third layer is the interpretation of the dynamics specified by the DEVS formalism. The fourth layer is reached by applying to the hierarchical abstract simulator a schema for synchronization and intercommunication among components. This fourth layer is called the distributed simulator. Finally by a mapping process, the distributed simulator is implemented on a hardware/software architecture.

In Concepcion [2], a design of a distributed simulator was proposed, the Hierarchical Multi-Bus Multiprocessor Architecture (HM^2A). This design can be readily implemented with off-the-shelf technology and directly reflects the abstract simulator specification. The architecture is designed around a primitive which is a cluster of processing elements communicating via a local bus and each cluster communicates via inter-cluster bus. Mapping the hierarchical abstract simulator onto the proposed architecture was shown to be a straight forward recursive manner [3].

This paper discusses the fourth step in the distributed simulation methodology, mapping the hierarchical abstract simulator onto a hardware/software architecture. This step serves as a convenient starting point in studying a variety of alternative physical simulator implementations. Also this paper presents an alternative realization of the hierarchical abstract simulator by using DENELCOR's FORTRAN 77, an extended FORTRAN for parallel programming, on the Heterogeneous Element Processor (HEP) computer. Section 2 reviews the dynamics of the hierarchical abstract simulator and its algorithms while in section 3, the translation of the algorithms to DENELCOR's FORTRAN 77 is discussed. Section 4 presents the performance (execution time) of the implemented hierarchical abstract simulator on the HEP computer. Finally, section 5 proposes future directions on this work.

2. Hierarchical Abstract Simulator

The hierarchical abstract simulator is an intermediate state in realizing the model on a physical implementation of the distributed simulator. The hierarchical abstract simulator consists of a network of coordinators where each controls a set of subordinates. If a subordinate is also a coordinator, then it too controls a set of subordinates, and so on. A subordinate which is not a coordinator is called a simulator. The algorithms for the hierarchical abstract simulator define the procedure in computing the state of the DEVS component, updating the simulation time and scheduling new events.

Six types of messages were identified in [2] as sufficient for current execution of DEVS models: (x,τ), $(*,\tau)$, (o,τ), (y,τ), *done* and t_N respectively, these carry external event information, internal event notices, output information, processor termination and next event information. In this paper, the (o,τ) message is not included in the implementation. The (o,τ) message is used to increase the degree of parallelism in the hierarchical abstract simulator when several simulators have output available from the last computation. These messages are exchanged among the coordinators in the interior and root of the hierarchical structure and the workhorse simulators at its leaves. The routing tables and code schemes for the coordinators and the process descriptions for the simulators were specified in terms of functional units to facilitate their realization at the implementation layer. The resulting logical structure was shown to be a correct imple-

mentation scheme, and to be free of interferences and deadlocks [2]. This contrasts with other approaches which attempt to maximize parallelism by loosening up on the strict timing requirements of simulation. These approaches must necessarily allow for rolling back the simulation when an out-of-sequence event is detected. In summary, our approach aims for simplicity and uniformity of design, with guaranteed deadlock prevention.

Procedurally, the algorithms that describe the dynamics of the hierarchical abstract simulator are given in Figures 1 and 2. Note that each algorithm is guarded by a lock/unlock operation to assure mutual exclusion.

The following is a list of variables used in the algorithms:

t_L = time of last event.
τ = global time.
t_N = time to next event.
ta = time advance function.
i^* = immiment component (minimum t_N).
e = elapsed time in this state.
s = state of the model component.
δ_{ext} = external transition function.
δ_{int} = internal transition function.
y = output from model component.
λ = output function.
(x,τ) = input external message x occuring at time τ.
$(*,\tau)$ = input internal message occuring at time τ.
(y,τ) = output message occuring at time τ.
EXT_IF TABLE = external interface table.
INT_IF TABLE = internal interface table.
OUT_IF TABLE = output interface table.
MINTN = function that determines the minimum t_N.

There are two groups of algorithms, one for a coordinator and one for a simulator. Each group is divided into: *when receiving an* (x,τ) *message* and *when receiving an* $(*,\tau)$ *message*. The following gives a summary of the actions taken by the components of the hierarchical abstract simulator when receiving a message.

When a simulator receives an $(*,\tau)$ message: it checks first the simulation time t, then it sends its output as (y,τ) to its coordinator. Simultaneously, the simulator computes its new state which includes determining a new t_N which is sent to the coordinator. At termination of computation, the simulator sends its *done* signal.

```
1.  when receive an input (x,τ):
2.     lock (bit)
3.     done := false
4.     if t_L ≤ τ ≤ t_N  then
5.          [ e:= τ - t_L
6.          s:= δ_ext(s,e,x)
7.          t_L := τ
8.          t_N := t_L + ta(s)
            ]
9.     else error
10.    done := true
11.    unlock (bit)
12.  end when receive
```

(a) Algorithm when receiving a (x,τ) message.

```
1.  when receive an input (*,τ):
2.     lock (bit)
3.     done := false
4.     if τ = t_N  then
5.          [ cobegin
6.               y:= λ (s)
7.               send (y,τ) to coordinator
8.               s:= δ_int(s)
9.            coend
10.         t_L := τ
11.         t_N := t_L + ta(s)
            ]
12.    else error
13.    done := true
14.    unlock (bit)
15.  end when receive
```

(b) Algorithm when receiving a $(*,\tau)$ message.

Figure 1: Algorithms for a Simulator

```
1.  when receive an input (x,τ):
2.     lock (bit)
3.     done := false
4.     if t_L ≤ τ ≤ t_N  then
5.          [ send input (x_i,τ) to all the affected
              simulators i via a  table look-up of
              EXT_IF TABLE
6.          wait until all simulators i's done are true
7.          t_L := τ
8.          t_N := MINTN(all subordinates under
                 coordinator)
            ]
9.     else error
10.    done := true
11.    unlock (bit)
12.  end when receive
```

(a) Algorithm when receiving a (x,τ) message.

```
1.  when receive an input (*,τ):
2.     lock (bit)
3.     done := false
4.     if τ = t_N  then
5.          [ send the input (*,τ) to i*
8.            when receive an input Y sent by i*
9.               :Y is used in the same enclosure:
10.                   send the message (x,τ) to each i*
                      influencees via a table look-up
                      of INT_IF TABLE
11.                   wait until i* influencees' done
                      are true
12.              :Y is used outside of the enclosure:
13.                   send the message (y,τ) to the
                      next level coordinator via a
                      table look-up of OUT_IF TABLE
14.              :wait until i* done is true
15.           end when receive
23.         t_L := τ
24.         t_N := MINTN(all subordinates under
                 coordinator)
            ]
25.    else error
26.    done := true
27.    unlock (bit)
28.  end when receive
```

(b) Algorithm when receiving an input $(*,\tau)$ message.

Figure 2: Algorithms for a Coordinator

When a coordinator receives a $(*,\tau)$ message: it checks the simulation time, t, then it sends the $(*,\tau)$ message to the component with the minimum t_N. This component is called the imminent component. The coordinator then waits for *done* signal from the imminent component. Afterwhich the coordinator proceeds to determine the new imminent component.

When a simulator receives an (x,τ) message: it checks the simulation time first and then it computes its new state, s. A new t_N is determined which is sent to the coordinator. At termination, the simulator sends *done* signal to the coordinator.

When a coordinator receives an (x,τ) message: it performs a check on the simulation time and then it sends the reformatted (x,τ) message to the affected subordinate by a table look-up of EXT_IF TABLE. The coordinator waits for all affected components to send *done* signals. Afterwhich the coordinator proceeds to determine the new imminent component.

When a coordinator receives a (y,τ) message from its subordinate, it determines whether this message is used within its enclosure or not. If the message is used within, then the coordinator sends the (y,τ) message as an (x,τ) message to the affected subordinate by a table look-up of the INT_IF TABLE otherwise, by a table look-up of OUT_IF TABLE, the coordinator sends the message (y,τ) to the next higher level coordinator.

3. Implementation on the HEP

The architecture of the Heterogeneous Element Processor (HEP) has been described in [6,7]. As shown in Figure 3, the main components are the Data Memory Module, the Packet Switch Network and the Process Execution Module. A program consists of one or more tasks while each task consists of one or more processes. Each process is composed of a sequence of instructions. Both the tasks and processes are executed in parallel in the HEP while the instructions of each process are executed in sequential pipeline fashion. Each PEM has a program memory where active tasks and processes instruction streams are selected for execution. Up to 50 instruction streams can be active at any given time. Notice that each PEM has a number of functional units which allow pipeline execution of multiple instruction streams for multiple data streams. This makes the HEP computer an MIMD machine.

For software support, HEP has the DENELCOR's FORTRAN 77 [5]. It provides the parallel programming environment for the HEP computer. It generates fully reentrant (sharable) code and provides synchronization among these codes. As shown in Figure 4, CREATE commands initiate processes A, B and C which execute in parallel with the MAIN. Synchronization among these processes is done via F/E (full/empty) bit that is tagged on special shared variables called asynchronous variables. These variables are prefixed with a "$" character. The following are some of the functions of the asynchronous variables:

J = $I, wait for full and set empty (integer).

X = $A, wait for full and set empty (real).

$Y = B, wait for empty and set full.

A = WAITF($B), wait for full, but do not set empty (real).

L = EMPTY($Q), test for empty access state

A = VALUE($Q), read regardless of state and leave unchanged (returns logical result).

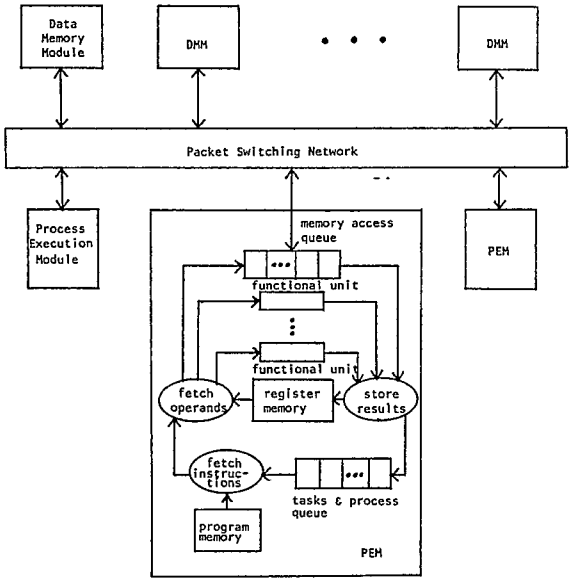

Figure 3: The HEP Functional Organization

For process initiation,

CREATE MYSUB(X,Y,Z), causes referenced subroutine MYSUB to execute in parallel with the creating routine with parameters X, Y and Z.

RETURN, terminates the parallel process executing a subroutine that was CREATEd.

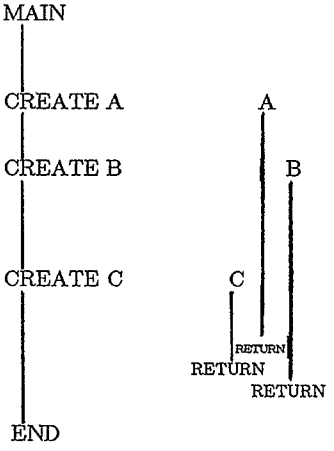

Figure 4: Process Initiation and Termination

Shown below is an example of parallel program in DENELCOR's FORTRAN 77 which creates four parallerocesses and performs all four executions of the subroutine S concurrently.

```
C     MAIN PROGRAM
      COMMON $NP
      PURGE $NP
      $NP = 4
      CREATE PS(1)
      CREATE PS(2)
      CREATE PS(3)
      CALL PS(4)
   20 IF (VALUE($NP).NE.0) GOTO 20
      END
      SUBROUTINE PS(N)
      COMMON $NP
      CALL S(N)
      $NP = $NP - 1
      RETURN
      END
```

The subroutine S is reentrant and the $NP is an asynchronous variable which is used to record the number of processes still executing. When S is finished, $NP is decremented. The MAIN PROGRAM waits until the value of $NP is zero. This means that all processes have finished executing S.

The implementation of the hierarchical abstract simulator on the HEP computer consists of translating the algorithms for a coordinator and for a simulator (see Figures 1 and 2) into the DENELCOR's FORTRAN 77. As seen from these algorithms, there are five types of messages, excluding (o,τ) message, being transmitted: (x,τ), $(*,\tau)$, (y,τ), t_N and *done* messages. The implementation is currently restricted to a binary structure with a maximum of 3 levels. The implementation can be easily expanded to more than 3 levels and applicable to general tree structure. At 3 levels, the hierarchical abstract simulator consists of three coordinators, see Figure 5:

- C_0, C_1 and C_2

and four simulators:

- $S_{1.1}$, $S_{1.2}$, $S_{2.1}$ and $S_{2.2}$

Also there is a process called GEN which generates the messages (x,τ) and $(*,\tau)$.

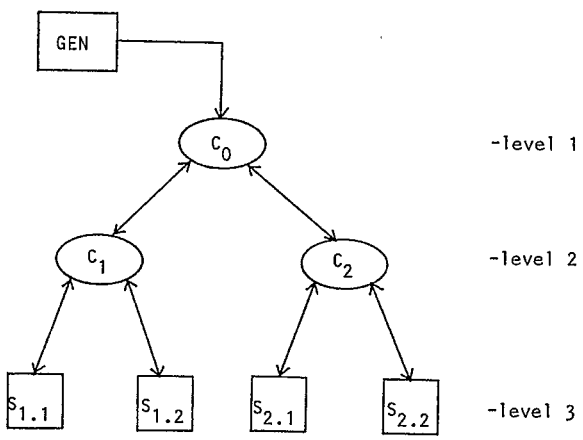

Figure 5: The Hierarchical Abstract Simulator Implemented on the HEP Computer

The main program does the following functions:

1. Obtain from the user the desired assignment of processors and other initialization inputs.
2. CREATE or CALL the processes for the appropriate coordinators and simulators.
3. Initialize the state and control variables of each process CREATEd or CALLed.
4. Perform the function of GEN and test for the termination of the execution.

The following are the inputs to the hierarchical abstract simulator at initialization:

- Specify whether the trace for debugging will be turned off or on.
- Specify the assignment of processors to the 3 leveled hierarchical abstract simulator. This is done by an input string of 7 bits. A 1 in this string means a processor is assigned, a 0 means no processor is assigned. The positions of the bit string corresponds to the list $C_0 C_1 C_2 S_{1.1} S_{1.2} S_{2.1} S_{2.2}$.
- Enter the desired percentage of $(*,\tau)$ messages of the total messages generated by GEN, e.g., entering a 40 means that an average of 40% of the generated messages will be of $(*,\tau)$ type.
- Enter the total number of messages to be generated by GEN. The execution terminates when there are no more messages to be processed.

All the CREATEd processes at initialization are passive except the process assigned to GEN. When GEN produces the first message, the execution of the distributed simulation begins.

The following are the variables used for message passing and synchronization:

1. $MESS(*process id*), this is an array of asynchronous variables indexed by the process id. Each element in this array contains either an (x,τ) or $(*,\tau)$ message. The receipt of this message signals the process (either coordinator or simulator) to begin executing the appropriate subrotine. For each process, the following statement

   MYMESS = $MESS(*process id*)

   will force the process to wait if the right hand side is *empty* or to continue executing if the right hand side is *full.*
2. $DONE(*process id*), this is an array of asynchronous variables indexed by the *process id.* An element in this array contains the signal to a coordinator that a subordinate with the index *process id* has completed its execution. If process α is busy computing then $DONE($\alpha$) is full, otherwise it is empty.
3. TL(*process id*), this is an array that contains each process' time of last event, t_L.
4. TN(*process id*), this is an array that contains each process' time to the next event, t_N.

For a coordinator, the statement

MYMESS = $MESS(*process id*)

is used to determine whether a message was sent either by another coordinator or a subordinate.

The statement

$MESS(*process id*) = MYMESS

is used to send a message to a process (a coordinator or a simulator).

The statement

$$\$DONE(\textit{subordinate process id}) = 1$$

is used to flag the subordinate process to be in busy state. This means that the subordinate is busy computing by setting the asynchronous variable $DONE full.

Then the statement

20 IF(EMPTY($DONE(*subordinate process id*)).EQ.FALSE)GOTO 20

causes the coordinator to wait until the subordinate process is finished computing.

For a simulator, the statements

$$\text{MYMESS} = \$MESS(\textit{process id})$$
$$\$MESS(\textit{coordinator process id}) = \text{MYMESS}$$

are used to receive and send messages respectively.

The statement

$$\text{FINISH} = \$DONE(\textit{process id})$$

sets the asynchronous variable empty, thus signaling the appropriate coordinator that a subordinate has finished computing.

The computations of the following functions are simulated by holding the process for a randomly selected duration of time:

- δ_{int}, internal transition function.
- δ_{ext}, external transition function.
- ta , time advance function.
- λ , output function.

Thus we have a system of concurrent processes where there is no assumption made on the order of processes finshing their computations of state variables.

4. Experimental Runs and Results

As mentioned in section 3, the implementation of the hierarchical abstract simulator consists of 3 levels with 3 coordinators and 4 simulators. The advantage offered by the hierarchical abstract simulator is the exploitation of the parallelism inherent in the model, i.e., the external events sent by a model component to its influencees can all be processed concurrently. The parallelism is facilitated by the hierarchical model decomposition and such parallelism may thus grow exponentially with the number of levels of a hierarchical DEVS model.

Unfortunately, such gains from parallelism cannot be fully realized. Experimental runs were made and three factors were found to affect the execution time of the implementation on the HEP computer:

(a) Constraints of the hardware architecture (number of processors).
(b) Frequency of synchronization and intercommunication.
(c) Workload (number of messages to be processed).

This section presents the effects of the above factors on the execution time of the implemented hierarchical abstract simulator. With regards to the constraints of the hardware (number of processors), we run the following assignments of processors:

- 3 processors, with each coordinator being assigned a processor and no simulators being assigned a processor.
- 4 processors, each coordinator is assigned a processor and one of the simulators is assigned to a processor.
- 5 processors, each coordinator is assigned a processor and two of the simulators are assigned each with a processor.
- 6 processors, each coordinator is assigned a processor and three of the simulators are assigned each with a processor.
- 7 processors, the full assignment.

When there are not enough processors assigned, the processes share processors which forces them to execute in a sequential manner. Only the last configuration has a one-to-one assignment.

The frequency of synchronization and intercommunication is simulated by varying the percentage of $(*,r)$ to (x,r) messages that is generated by GEN. Also runs were made for processing 500 messages compared to 1000 messages.

Several runs were made of the hierarchical abstract simulator implementation and the results are summarized in Figures 6 and 7. Shown in Figure 6 is the effect on execution time by varying the percentage of $(*,r)$ messages. The more $(*,r)$ messages in the system, the more intercommunication occurs. This is due to the generation of (y,r) messages by the simulator when it receives a $(*,r)$ message, see Figure 1(b). The (y,r) message is routed to its destination by the coordinator either within or outside of its enclosure, see Figure 2(b). An increase in the number of $(*,r)$ message processed by the hierarchical abstract simulator, the longer is the execution time. A decrease in execution time is noted when there are

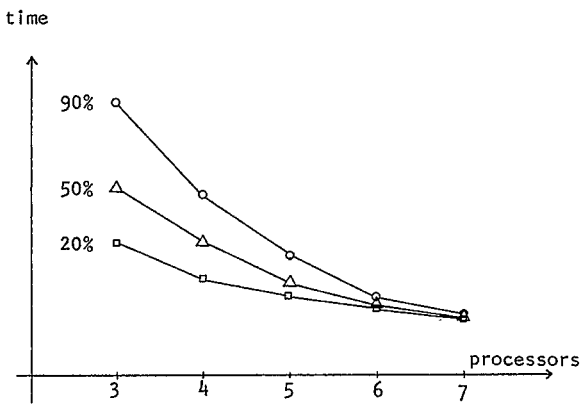

Figure 6: Runs Made for Changing Percent of $(*,r)$

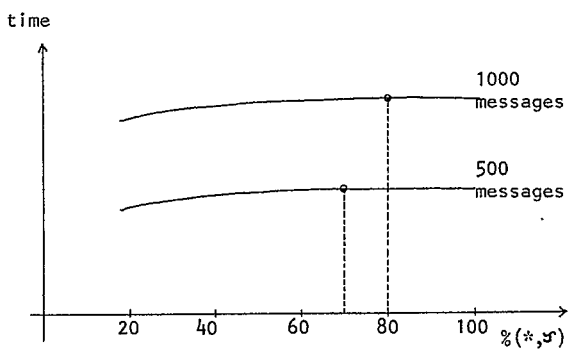

Figure 7: Runs Made for Changing the Number of Messages (using 7 processors)

more processors assigned to the hierarchical abstract simulator but this decrease is not so significant from 6 to 7 processors. Because of the under utilization of some of the processors, the gain by using one more processor (from 6 to 7) is not fully realized.

To get some insight on the effect of increasing the number of messages to be processed, runs were made and the results are shown in Figure 7. This result was obtained by using the full assignment of processors, which is 7. As expected, there is an increase of execution time when the hierarchical abstract simulator is processing more messages. But it was also observed that at 500 messages, the execution time did not increase beyond 80% (*,t) and at 1000 messages, the peak is reached at around 70% (*,t). This shows a saturation point where the increase of overhead due to intercommunication did not affect the execution time. This is due to the parallelism inherent in the hierarchical abstract simulator, the increase of intercommunication is absorbed by the concurrent execution of the processors.

Runs were also made to determine the effect of the following routing characteristics of messages:

- Having more (x,τ) messages routed to both subordinates of a coordinator.
- Having more (y,τ) messages routed to a subordinate and to the next higher level coordinator.

The first characteristic simulates the occurrence of having more simulators affected by an external message, (x,τ). This results in more concurrency in the execution of simulation. The second characteristic simulates the sending of output messages to the most remote simulator. This occurs when the (y,τ) message has to be sent by a coordinator to the next higher level coordinator. The results did not show any significant difference of execution times for both characteristics. This is due to the fact that the coordinator has no delays in doing the following activities:

- table look-up, to determine the destination of the message via the interface tables.
- determining the minimum t_N, the function MINTN was performed in 0 processing time.

But significant difference in execution times were observed when using different assignments of processors. The full assignment sometimes shows half the execution time compared to the execution time for 3 processors.

5. Conclusion

This paper has shown an alternative implementation of the mapping of the hierarchical abstract simulator to a hardware/software architecture. The HEP computer with its MIMD architecture and the support of a high level parallel language, DENELCOR's FORTRAN 77, the combination produces a very viable implementation for distributed simulation. Although there is a great need for more disgnostics and debugging tools to trace and debug parallel programs.

Performance in terms of execution times were measured on different runs of the implementation. It was observed that the number of processors, frequency of synchronization and intercommunication, and number of messages affect the execution time.

The following gives a summary of the results obtained:

- that an assignment of processors close to the full assignment gives almost the same execution time as a full assignment.
- that the execution time increases when there are more (*,τ) messages than (x,τ) messages to be processed.
- that for an assignment of processors, there is a saturation point where increasing the (*,τ) messages did not increase the execution time.
- that the execution time increases when there are more messages, (*,τ) and (x,τ), to be processed.

Some research have been done on performance of distributed simulation. Livny [8], discusses a measurement called the Optimal Execution Time which gives a relationship between the inherent parallelism and the number of concurrent simulators. Davidson and Reynolds [4] found out in their experiments of using 3 microcomputers for distributed simulation that the degree of communication degrades the performance of the simulators. The processes communicate with each other at the end of a certain time interval. If this time interval is less than 10 units of time, then degradation of performance was observed. In Baik and Zeigler [1], a methodology is presented for the performance evaluation of hierarchical distributed simulators. Their methodology measures the minimum average run time per task and the maximum throughput per unit of hardware complexity.

The difference of the above research from this work is that, an implementation of the distributed simulator is done on an actual multiprocessor architecture and that actual CPU real-time are measured. The results of this work also shows a saturation point for the hierarchical abstract simulator and that the full assignment of processors does not always produce the optimal performance.

Future work on the hierarchical abstract simulator implementation on the HEP computer will consists of the following:

(a) Inclusion of the (o,τ) message type and introducing delays in each coordinator for table look-up and MINTN activities.

(b) Expanding the current implementation to a general tree structure.

(c) Running a real-time simulation of a distributed computer system.

Acknowledgement

The author would like to thank Ann Hayes, Computing Research and Application Group, Los Alamos National Laboratory, for allowing us computing access to the HEP computer on site of the Los Alamos National Laboratory. The work would not have been possible also without the support of DENELCOR's consultants on site, Olaf Lubeck and Dale Carstensen. They have been very patient in answering our questions.

References

[1] Baik, D-K and Zeigler, B.P., "Performance Evaluation of Hierarchical Simulators", In Proc. of 1985 Winter Simulation Conference, San Francisco, CA, Dec 1985.

[2] Concepcion, A.I., "Distributed Simulation on Multiprocessors: Specification, Design and Architecture", Ph. D. Dissertation, Tech. Rep. CSC85-001, Dept. of Computer Science, Wayne State University, Jan 1985.

[3] Concepcion, A.I., "Mapping Distributed Simulators Onto the Hierarchical Multi-Bus Multiprocessor Architecture", In Proc. of the 1985 MultiConference: Distributed Simulation, San Diego, CA, Jan 1985, pp. 8-13.

[4] Davidson, D.L. and Reynolds, P.F., "Performance Analysis of a Distributed Simulation Algorithm Based on Active Logical Processes", In Proc. of 1983 Winter Simulation Conference, Arlington, VA, Dec 1983, pp. 266-268.

[5] DENELCOR, "FORTRAN 77 Reference Manual, Release 1.0", Publication No. 9008020-000, DENELCOR INC., 17000 E. Ohio Place, Aurora, Colorado, Jun 1984.

[6] Gajski, D.D. and Peir, J-K, "Essential Issues in Multiprocessor Systems", Computer, Vol. 18, No. 6, Jun 1985, pp. 9-27.

[7] Hwang, K. and Briggs, F.A., "Computer Architecture and Parallel Processing", McGraw Hill Book Company, New York, 1984.

[8] Livny, M., "A Study of Parallelism in Distributed Simulation", In Proc. of 1985 MultiConference: Distributed Simulation, San Diego, CA, Jan 1985, pp. 94-98.

[9] Zeigler, B.P., "Multifacetted Modelling and Discrete Event Simulation", Academic Press, London, 1984.

[10] Zeigler, B.P., "Discrete Event Formalism for Specification of Hierarchical Models", In Proc. of the 1985 MultiConference: Distributed Simulation, San Diego, CA, Jan 1985, pp. 3-7.

ARTURO I CONCEPCION received the B.S. degree in Mechanical Engineering from the University of Santo Tomas, Manila, Philippines, in 1969, the M.S. degree in Computer Science from Washington State University, in Pullman, in 1981, and the Ph. D. degree in Computer Science from Wayne State University, Detroit, Michigan, in 1984. Since 1982, he has been involved with research, funded by the National Science Foundation, on the theory, design and implementation of distributed simulation. He is currently an Assistant Professor in the Department of Computer Science, Michigan State University. He is now involved in a research group which studies the distributed control, efficiency and reliability of distributed computer systems. His principal interests are in distributed operating systems, networks, distributed databases, and modelling and simulation. He is a member of the ACM, IEEE-Computer Society and Sigma Xi.

Department of Computer Science
Michigan State University
E. Lansing, MI 48824
(517) 355-2359

Proceedings of the 1985 Winter Simulation Conference
D. Gantz, G. Blais, S. Solomon (eds.)

TWO APPROACHES TO THE IMPLEMENTATION OF A DISTRIBUTED SIMULATION SYSTEM

Murali Krishnamurthi
Industrial Automation Laboratory
Dept. of Industrial Engineering

Usha Chandrasekaran
Laboratory for Software Research
Dept. of Computer Science

Sallie Sheppard
Laboratory for Software Research
Dept. of Computer Science

Texas A & M University
College Station, TX 77843

ABSTRACT

This paper describes two approaches to the implementation of distributed simulation currently being pursued at Texas A&M University. The first approach describes the design and the implementation of a distributed simulation system onto a Motorola 68000 based architecture. This approach involves transparently distributing the language support functions of an existing simulation language (GASP) onto multiple processors. The second approach discusses the implementation of simulation support software in a high level distributed processing language. This approach involves the distribution of protions of the simulation model which can be executed in parallel onto multiple processors by the model builder. The paper discusses the details of both the approaches and the current status of their implementation.

1.0 INTRODUCTION

Since 1983 Texas A&M University has been involved in a project to design and implement a distributed simulation system. Funded in part by the National Science Foundation [31], the first phase of the project which ended in May 1985 concentrated on exploring software implementation strategies for distributed simulation [15,16,33,34]. Three strategies for the distribution of the software onto multiple processors were defined and emulated via multitasking on single processor systems. As a result of this work two of the strategies were selected for further study in the implementation phase of the project currently in progress. The first strategy involves taking an existing simulation language, GASP IV, and transparently distributing the support subroutines onto the available processors. All user-written model code is executed on a single processor thus avoiding problems in deadlock detection and avoidance. Various support functions such as random variate generation, statistics processing and filing are distributed onto the different processors. The second approach implements simulation support software in a high level distributed processing language. Here the distribution is not transparent to the model builder who must designate which portions of the model can be executed in parallel. This further means that the implementation must include provision for automatic deadlock detection and prevention but offers more potential speed-up from the distribution than the first approach.

The goal of current phase of the distributed simulation project is to construct hardware/software systems utilizing multiple processors to support simulation for both of these strategies. The two different designs being pursued dictate two different approaches in the implementation. The first design is being implemented on a distributed architecture of Motorola 68000 processors while the second design is being implemented in the Ada* programming language and will be portable to any distributed architecture supporting Ada. This paper describes these approaches along with the relative merits of each.

2.0 THE DESIGN AND IMPLEMENTATION OF A LANGUAGE SUPPORTED DISTRIBUTED SIMULATION SYSTEM

One approach to distributed simulation implementation is through the distribution of simulation language functions onto individual processors [5,6,33,36,37]. The basic difference between the distributed simulation via model function approach [2,4,26,30] and this approach is that the model function approach distributes simulation model functions onto separate processors whereas this approach distributes language support functions onto individual processors, thus exploiting the inherent parallelism in the language functions. This approach has the advantage of avoiding deadlock problems but the disadvantage of not exploiting any of the parallelism in the system being simulated.

One of the distributed simulation systems currently being built and implemented at Texas A&M University is based on the distributed simulation via language functions approach. The objectives of this system are (1) to implement a distributed simulation system using off-the-shelf hardware components, (2) to use an existing simulation language in the system, (3) to maintain the existing language and execution structures of the language, (4) to maintain the distributed implementation transparent to the user, and (5) to speed up the simulation at a low cost. The following sections describe the design, architecture and the implementation status of this system.

2.1 System Design

Designing a dedicated system to support distributed simulation necessitates a clear definition of the requirements. For example, requirements such as the type of architecture, the

* Ada is a trademark of the U.S. Department of Defense.

This material is based upon work supported in part by the National Science Foundation under Grant No. ECS-8215550

type of interprocessor communication, the type of operating system configuration and the language to be used have to be defined. Generally, the multiprocessor architectures are classified by the interconnection structure between the processors and the memories. The three most common interconnections are (1) time shared or common bus, (2) cross bar switch network, and (3) multiport memories [10]. Among the three, the common bus interconnection scheme is the least expensive and the least complex scheme, but it is also the least efficient scheme. The common bus interconnection scheme is ideal for building dedicated, exploratory multiprocessor systems using off-the-shelf hardware components since the hardware complexity is minimal in this scheme. Several distributed simulation systems have been designed based on the common bus architecture or on the enhancements of the same [25,27]. The common bus architecture has been chosen for this system since it is the simplest of the interconnection schemes and also because off-the-shelf hardware is available for this type of interconnection scheme.

Since one of the objectives of the distributed simulation system is to use an existing simulation language, the GASPIV Simulation Language [28] has been chosen for implementation (see [38] for the reasons for selecting GASPIV). The selection of GASPIV required the analysis of the language to identify major functional subprogram groups which could be shown to demonstrate relative independence during execution. A complete analysis of GASPIV showed that the subprograms could be grouped into eight tasks which are mutually exclusive for most of the simulation run except during program initialization and termination. Since these tasks are mutually exclusive they have independent instruction streams and have been partitioned into eight separate tasks with each partition executed on a separate processor. Figure 1 shows the eight partitions and the subprograms grouped within each partition.

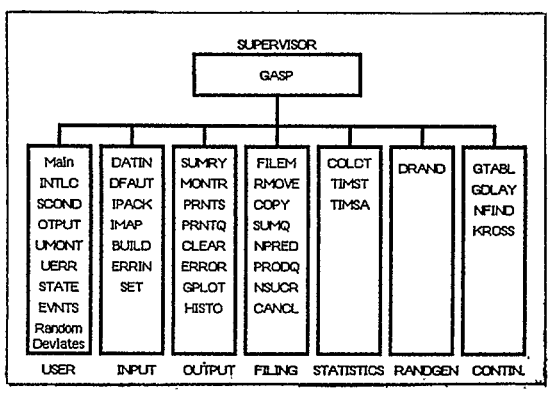

Figure 1. GASPIV Subprogram Paritition Groups

Even though the eight tasks are mutually exclusive in terms of processing activity, discrete simulation requires the tasks to communicate with one another to exchange the necessary results. This requires designing a mechanism to allow the tasks executing on separate processors to communicate with one another. There are several mechanisms available for interprocessor communication in multiprocessor systems [18]. In the case of the common bus architecture it is possible to design a tightly coupled system with the processors communicating through shared memory or a loosely coupled system with the processors communicating directly through the bus. It is also possible to design a system with a single bus or with multiple buses [7] based on the system requirements and the availability of off-the-shelf hardware to meet those requirements. A single common bus architecture with shared memory type communication or a common bus architecture with interrupt driven communication mechanism are the most commonly used architectures for distributed simulation [27].

Since the simplest mechanism for FORTRAN tasks to communicate with one another is through a global common data area (BLOCK DATA), the shared memory type communication (with the global common area located in the shared memory) has been designed for this distributed simulation system. The tightly coupled architecture of the system with one supervisory task and seven slave tasks distributed on eight processors required a compatible operating system configuration. The commonly used operating system configurations in multiprocessing systems are the *master-slave*, *floating supervisor*, and *separate supervisor* type configurations. The master-slave type configuration has been chosen for this distributed simulation system since it is compatible with the hierarchical design of the system and also because it is the simplest of the operating system configurations available for multiprocessors built from off-the-shelf hardware components.

After the design of the distributed simulation system was completed its feasibility was verified by emulating the system on a Texas Instruments 990/12 minicomputer. The emulation provided satisfactory results on the feasibility of the designed system (see [15,16] for details on the emulation of the system).

2.2 Hardware Architecture

The design of the distributed simulation system necessitated that the processors should be capable of executing tasks of size at least 64K and allow the creation of sizable user programs. The architecture required that the processors should be capable of communicating through the common bus and the shared memory. The system design also required that the selected processors should have additional features such as an I/O bus to facilitate user interaction and communication with peripheral and storage devices, a suitable operating system and adequate software support.

The selection of the hardware depended on the availability of the hardware that satisfied the system requirements. Motorola's 16-bit microprocessors have been selected for the the distributed simulation system since they met the system requirements and were also relatively inexpensive compared to mini or mainframe computers. The hardware consists of a VME/10 microcomputer and seven VME110 monoboard microcomputers interconnected via a common bus called the VMEbus and associated hardware components such as serial ports, card cage and power supply. Figure 2 shows the configuration of the hardware as designed in this project. The following subsections describe the operation of the hardware and its configuration.

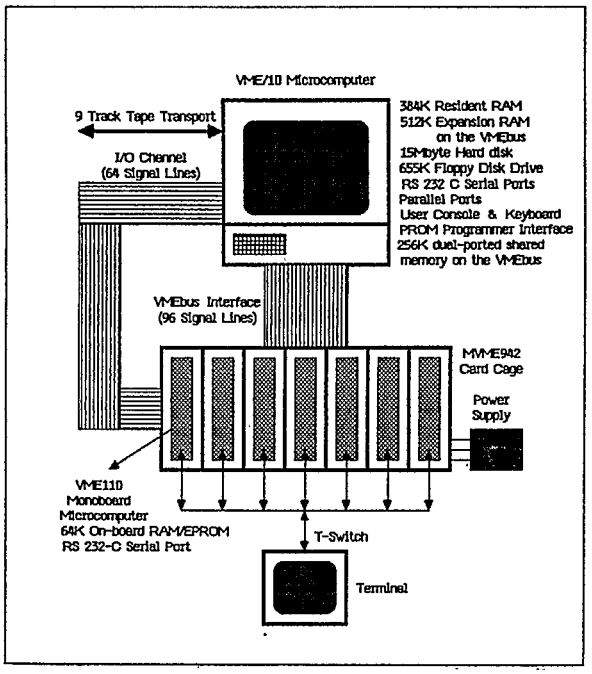

Figure 2.Hardware Setup of the Language Supported Distributed Simulation System

2.2.1 Hardware Description

The VME/10 is a development system consisting of a M68010 processor and the VERSAdos operating system. The VME/10 has a 15megabyte hard disk, a $5\frac{1}{4}''$ floppy disk drive, 384K of RAM expandable to 1152K and a 16 megabyte addressing range. The VME/10 has three bus facilities: a local on-board bus, an I/O channel (or an I/O bus) and the VMEbus. The local bus provides communication between the microprocessor unit, memory management unit, keyboard, RAM, ROM, CRT and the I/O channel. The I/O channel consists of 64 signal lines and interfaces the local bus to hard disk and communicates with off-board devices such as serial ports, parallel ports, terminals and printers. The VMEbus is an industry standard bus with 96 signal lines which allows the VME/10 to access additional memory, processors, or controllers. The VME/10 requires the configuring of its memory map, I/O ports, and the tailoring of its operating system at system generation to suit the customized hardware configuration of the system. The VERSAdos operating system is a multitasking, multiprogramming operating system which supports high level languages such as FORTRAN, Pascal and other software utilities (see [24] for more information on the VME/10 and the VERSAdos operating system).

The VME110 is a single board microcomputer that can function as a stand-alone microcomputer or as one of several CPU elements in a multi-processor VMEbus configuration [22]. The VME110 is a 16-bit microprocessor with an MC68000 processor, 64K on-board RAM/ROM/EPROM and a 16 megabyte addressing range. The VME110 has similar bus features as the VME/10 and it can also access off-board resources on the VMEbus.

The VMEbus interface on the VME/10 and the VME110 provides data and address path from the on-board MPU via the local bus to the VMEbus. The VMEbus interface system is comprised of four groups of signal lines called *buses* and a collection of *functional modules* which can be configured as required to interface devices to buses. The four buses are, (1) Data Transfer bus, (2) Data Arbitration bus, (3) Interrupt bus, and (4) Utility bus. The Data Transfer Bus (DTB) is used by the devices to transfer data and the DTB contains the data and address pathways and the associated control signals. Functional modules (a collection of electronic components with a single functional purpose) called *DTB Masters* and *DTB Slaves* use the DTB to transfer data between each other. The Data Arbitration Bus is used to guarantee that only one DTB Master is in control of the bus at any time since it is possible to configure the VMEbus with several DTB Masters. The Data Arbitration Bus is used to transfer control of the bus between DTB Masters and this is performed by the modules DTB Requester and the DTB Arbiter. The Interrupt Bus facilitates the interruption of the normal bus activity by devices so that the interrupt requests can be serviced. The interrupt requests can be prioritized to a maximum of seven levels. The functional modules associated with the interrupt bus are the Interrupters and the Interrupt Handlers which use the signal lines of the interrupt bus. The Utility bus includes a collection of utilities for failure detection, system clock, initialization and system reset (see [11,12,35] for more information on the VMEbus and its specifications).

2.2.2 Hardware Configuration

The use of off-the-shelf hardware in the distributed simulation system requires configuring the hardware, integrating all the hardware components and customizing the operating system to suit the desired system design. The VME110 processor, when supplied contains only the processor, bus interfaces and the basic hardware components. The memory devices, the address map decoder and the operating system are not provided with the processors since they have to be selected and configured as required by the application system. The memory map of the VME110 has to be configured to allow the accessing of on-board RAM/ROM/EPROM, off-board RAM (shared dual-ported memory accessible through the VMEbus) and the on-board boot-strap software. The configuring of the dual-ported memory accessible through the VMEbus as off-board memory for both the VME/10 and the VME110s enables the processors to share the memory for communication purposes. After the memory map had been appropriately configured, the address map for the memory access was designed and programmed into an address map decoder PROM and installed on the VME110. The operating system for the VME110 was generated from the VERSAdos utilities and the necessary device drivers available on the VME/10. The customized operating system was then programmed into EPROMs and installed on the VME110 (see [17] for more information on the hardware configuration of the distributed simulation system).

The integration of the system involved the interconnection of the various hardware components, the establishment of hierarchical control levels in the system, the establishment

of user interface and the implementation of the software. The VME110 processors were interconnected with one another by housing them in a card cage with the VMEbus backplane [23] and the I/O channel. The integration of the VME/10 and the VME110 processors required the interconnection of the VMEbus and the I/O channel between the VME/10 and the VME110s card cage. The establishment of hierarchical control levels in the system required configuring one of the processors as the System Controller. The system controller provides system management and control functions to the distributed simulation system. The software architecture and its implementation are described in Section 2.3. User interface is necessary only to the VME/10 since the user program creation and simulation initiation and termination take place on the VME/10. The user is not required to interact with the VME110 processors since the execution of the language tasks on the slave processors is maintained transparent to the user.

2.3 Software Architecture

Since the system objectives included maintaining the existing language structure of GASPIV and its user interface, a unique design of the software architecture was required. A software kernel was built around the GASPIV language tasks to allow the subprogram groups to execute independently and communicate with each other. In addition a software layering approach was developed to maintain the existing functional flow of GASPIV and its user interface. This software architecture is described in the following subsections.

2.3.1 Software Design

In GASPIV, the user writes the program, event routines, system initialization routines and any other necessary routines. When the user's main program is executed, it calls the subroutine GASP which establishes the simulation environment. From then on, subroutine GASP takes over until the specified completion time of simulation. After the completion of the simulation, subroutine GASP returns to the user's main program where the simulation may be terminated by the user's main program. The distributed simulation environment is required to maintain this conventional execution structure of GASPIV.

The implementation of the eight partitioned GASPIV language tasks in the distributed simulation system requires the consideration of these needs: (1) the partitioned language tasks containing subprograms written in FORTRAN need a MAIN program or a driver for each task (except the USER task which will be driven by the user's main program) to execute independently, (2) the subprograms need a mechanism to call subprograms residing in other tasks executing on separate processors, and (3) the need to interface user's programs with the other tasks. To satisfy these system needs a software layering approach has been developed. The software architecture of the distributed simulation system consists of three layers namely, (1)the GASPIV subprogram group, (2) the task interface layer which interfaces a subprogram group with other subprogram groups and the user programs, and (3) the operating system which allows the programs to access the common bus, shared memory and other system resources in the distributed simulation environment.

The inner layer contains the GASPIV subprograms in their original form. These subprograms residing on different tasks are interfaced with one another through the *task drivers* and the *task interface library*. The task driver is the main program of a task group which can execute the subprograms residing in its task at the request of a subprogram residing in another task and can suspend or terminate itself. The task interface library consists of *pseudo* subprograms of all the subprograms needed by more than one task. When a subprogram residing in a task calls another subprogram which is not residing in the same task, the pseudo subprogram of the called subprogram in the task interface library is referenced. The pseudo subprogram serves as a communication vehicle between the calling subprogram and the called subprogram. The pseudo subprogram places the subprogram parameters in the shared memory and sets the semaphore of the actual subprogram to be executed. The task containing the called subprogram detects this change in status of this semaphore in the shared memory and reads the subprogram parameters from the shared memory and executes the requested actual subprogram residing in it. A pseudo subprogram GASP has been designed for inclusion in the USER task and this pseudo GASP will serve as the task driver for USER task (see [15,16] for more information on the software design).

2.3.2 Software Execution Structure

Once the tasks have been installed on the appropriate processors and the necessary input files for the simulation have been created, the simulation can be started by executing the user program. When the user program is executed, the user's main program will call subroutine GASP. The subroutine GASP in the USER task is actually the pseudo GASP which will first execute an assembler program to allocate the shared memory to the task. Then all the shared variables and the semaphores will be initialized in the shared memory and the language tasks residing on individual processors will be activated separately. The pseudo GASP will then set the semaphore of the SUPERVISOR task to execute the actual subprogram GASP residing in it. The SUPERVISOR task will check its semaphore, detect the request for executing GASP and will execute subprogram GASP. The subprogram GASP will take over from here as in the conventional GASPIV execution. The interactions between the subprograms, task driver and the task interface library are shown in Figure 3. After the completion of the simulation, the SUPERVISOR task will send a message to all tasks except the USER task to terminate themselves and then terminate itself. The USER task will find that all the tasks have terminated from the change in their semaphore status and will return to the user's main program and will complete normally.

2.4 Current Status

The software development and the design verification phases of the language supported distributed simulation system have been completed. The processors and the other necessary hardware have been acquired and configured. The remaining tasks involve the completion of the software implementation and the testing of the system. The final phase of this project will involve the performance evaluation and the bench-marking of the developed system.

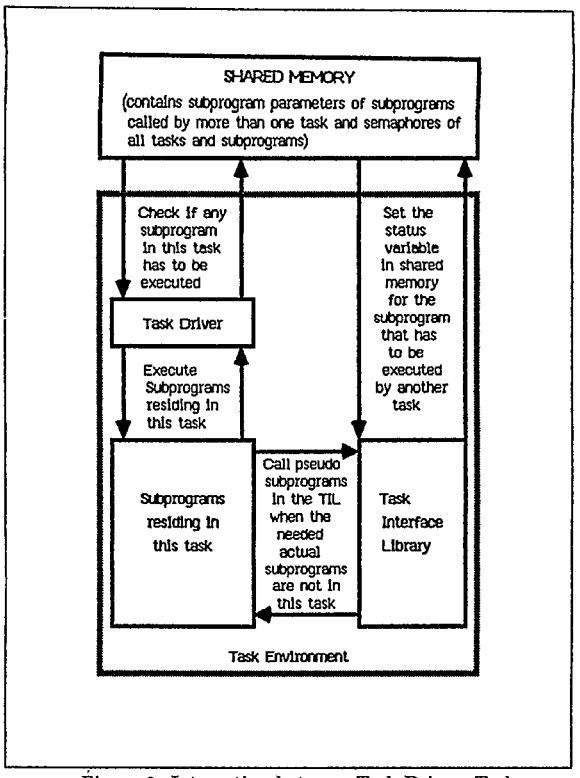

Figure 3. Interaction between Task Driver, Task Interface Library and the GASPIV Subprograms

3.0 IMPLEMENATION OF A MODEL BASED DISTRIBUTED SIMULATION

The principle behind distributed simulation is to introduce concurrency into the implementation so that the functionally independent units of the simulation model and the support functions can execute in parallel. The performance of such a system can be enhanced over that possible in the strategy described in section 2 by introducing concurrency into the components of simulation models themselves. This second approach is being researched at Texas A&M University. This effort explores the language requirements for distributed simulation of concurrent models. The objective of this research is to build the minimal simulation primitives suitable for distributed simulation on microprocessor architectures. Essentially the design includes an asynchronous simulation strategy, concurrent simulation primitives, deadlock prevention or recovery algorithms and a support environment. An overview of this approach is presented in this section.

3.1 Simulation Modeling Technique Suitable for Distributed Simulation

The simulation strategy determines the modeling methodology and the fundamental nature and world view of the system. Kiviat [14] identified three major modeling strategies in discrete simulation: (i) event scheduling, (ii) activity scanning, and (iii) process interaction. The event oriented methodology represents an instantaneous occurrence as an event and carries out the simulation by scheduling these events. The activity scanning approach carries out an action if the corresponding state changes and time scheduling conditions are met. The process interaction methodology models the system as a set of coexisting or cooperating processes each communicating through messages. Each process unit is controlled independently and the simulation is carried out by activity scanning or event scheduling.

From an analysis of the existing simulation strategies, the process interaction strategy was selected for the distributed simulation implementation since it maintains the inherent concurrency in the system being modeled to a greater extent than any other approach. The basic unit of computation is a process that sends and receives entities as messages: the entity flow between the processes characterizes the simulation progress. Thus the system to be simulated is modeled as a set of coexisting or cooperating processes. All processes execute concurrently and communicate through message passing interfaces. All messages or entities are time encoded and queued in transit. The message order is preserved between the processes and the time stamps of the messages are maintained in monotonically increasing order to insure proper and correct simulation.

3.2 Language Requirements for Distributed Simulation

The language requirements for distributed simulation can be broadly classified into three categories: power to express concurrent activities at source level, a distributed control mechanism to carry out simulation and a minimal set of modeling tools. These are described in the subsections below.

3.2.1 Distributed Simulation Control Mechanism

The system to be modeled is represented as parallel processes which operate on entities and send them to other processes through a message passing mechanism. Thus each process removes entities from its input message queue till it is empty or till the simulation termination conditions are satisfied, performs the necessary operations, updates its status and sends the entity or message to the next process in line. Figure 4 represents such a system with processes shown as nodes and message paths as arcs. Each node has a message buffer such as the one shown for P_4 which contains the time-ordered input messages for that node. A process with multiple input edges and messages on only a subset of them, like P_4 in Figure 4, has to wait until it has at least one message on all of them to simulate correctly. Such a process enters a *blocked* state. But this is overly restrictive since a blocked process with partial message input can still simulate forward without causing any incorrectness under certain conditions. The validity of the above statement is a direct consequence of the assumption that the messages have increasing time stamps along any virtual channel: in other words, a process can never send a message in its past. Thus P_1 cannot send a message with time stamp less than 110 units. Hence a receiving process can never receive a message with time stamp less than the minimum clock time of its predecessors and it can simulate or process the messages with time stamps less than or equal to the smallest local clock time of its predecessors. In Figure 4, all input edges of P_4 except the one between P_1 and P_4 have messages. The forward

simulation time of P_4 is the minumum of the clock values of P_1, P_2, P_3 and P_n and is 90. Thus P_4 can still process all the messages with time stamp less than or equal to 90 though it does not have a message from P_1.

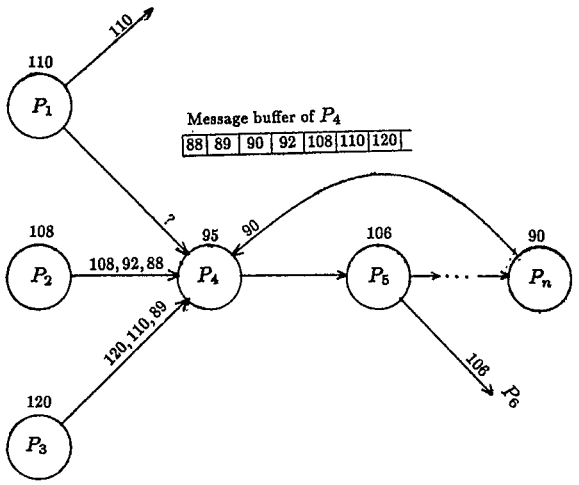

Figure 4. Blocking Situation

The basic principle behind the asynchronous execution of the simulation program without causing any incorrectness is to compute the safe forward simulation time (FST) for each process as the minimum of the local clock time of the predecessor processes and allow each process to operate on the messages with time stamp less than the safe forward simulation time. This algorithm is similar to the demand driven null messages method proposed by Chandy and Misra [4] except that the edges between processes do not have a clock associated with them. Rather, a successor process maintains and updates the clock value of its predecessors while processing the messages. Thus the update of the forward simulation time for a process is based on the clock value of its predecessors unlike the clock value of the edges as in the model proposed by Chandy and Misra. The advantage of this approach is that it avoids deadlock that arises due to total absence of messages along any edge. This situation is illustrated in Figure 5 in which P_5 keeps sending the messages to P_6 only. P_4 can not progress since its FST equals the local clock time of P_n. Hence P_4 would send an awakening signal to P_n requesting P_n to update its clock. This awakening signal is propagated to the predecessor P_k of P_n until the clock of P_k exceeds 90 units. If no such P_k exists the signal is transmitted back to P_4 which detects the deadlock situation and avoids by not considering the clock value of P_n in computing its forward simulation time. However, in the current situation, the clock value of P_5 namely 111 units will be sent to P_4 as reaction to the awakening signal which then can process all messages with time stamp less than or equal to 111 units. The reader is referred to [3] for further details.

Since the clock values are not maintained for the edges all the similar messages from various predecessors are enqueued in a single buffer. This approach also makes the handling of a multiple entity simulation system much easier. The same simulation strategy and the queueing algorithms can

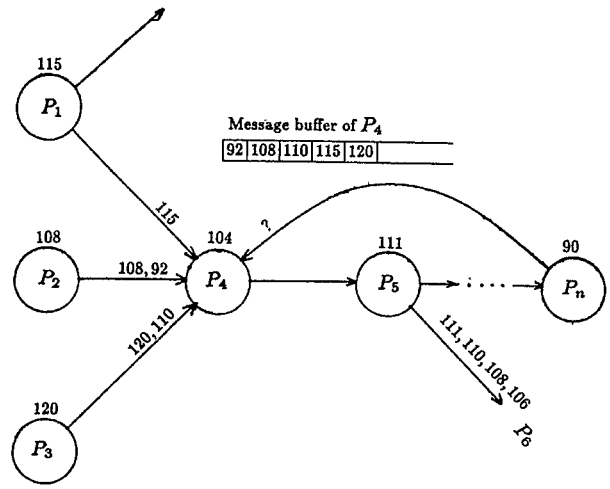

Figure 5. Deadlock Situation

be easily extended to simulate a multiple entity system in which processes send or receive more than one type of entity. Thus the number of buffers for a process is dictated by the different types of entities received by it and not by the number of edges between its predecessors and itself.

This strategy forms the crux of the run-time control environment and could be implemented as one single control module to govern the activities of all the user defined processes or as a set of concurrent control processes for each individual user defined processes. The second approach is a better alternative since the control module of each process conserves the locality and both the user defined process and its control module can be loaded onto the same processor in a multiprocessing environment. This approach is in accordance with the primary goal of developing a truly distributed simulation system.

3.2.2 Minimal Set of Modeling Tools

The modeler views a system to be simulated as a set of interacting processes that operate on the locally queued-in entities until the simulation termination conditions are met. Thus the modeling tools should have the following basic capabilities to build a simulation model: facility to represent and define the coexisting processes and entities of the real system, facility to create and remove an entity from the system, synchronized message communication mechanisms to simulate the flow of entities, access capabilities to the random number generators and statistics collection routines, and statements to begin and end simulation. This system is being built as an extension of a host language to allow rapid prototyping. The desired language features and the suitability of the chosen host language are discussed in the following section.

3.3 Language Features Essential for Distributed Simulation

The analysis of the languages suited for distributed simulation reveals that it should be able to handle the dynamic entity creation and queue handling. The number of entities prevalent in a system and the queue size of the

processes are dynamic during simulation. While languages like Ada [19] and Pascal provide access and pointer types to handle such dynamic situations, FORTRAN has to utilize static single dimensional arrays with predefined size. The shortcoming of using static arrays is that neither the model builder nor the system designer can estimate this parameter precisely due to the stochastic nature of the simulation problems. Furthermore this parameter will vary from problem to problem. While oversized arrays waste the memory space considerably, undersized arrays will jeopardize the simulation system performance.

The handling of entity flow has an impact on the simulation control environment. The entity flow can be handled by synchronized message communication, that is to transfer the entities with their attributes through the processes in the system or by storing the entities in a common global store and simulate the entity flow by sending a time encoded message. The first approach is ideal for a truly distributed architecture while the second approach needs a distributed architecture with a common global store, in a multiprocessing environment. However, the second method violates one of the operating characteristics of distributed systems namely not to have global variables and to use message passing protocols for all transfers, both in interprocess and interprocessor communications. Thus a communication mechanism like the rendezvous in the Ada programming language is ideal and necessary to represent the entity flow in a simulation system. The run-time system of the simulation language should also be capable of assigning the concurrent program units to different processors failing which the modeler should be provided with a facility to assign the concurrent program units to different processors. The entity definition along with its attributes, the operations to be performed on an entity like creation and destruction, queue handling mechanisms and the simulation termination conditions should be known at each individual processing unit to support distributed simulation. The simulation language also has to provide random number generators and statistics collection routines as concurrent units that emit a random number and accept an input data value respectively on a call from other program units.

Current research at Texas A&M University involves the rapid prototyping of the above mentioned concurrent simulation system. This implementation will provide the concurrent simulation primitives as extensions to a host language. The appropriate choice for the host language is a language with concurrent features at source level since it provides a natural base for the simulation implementation that has to support logically concurrent activities and synchronization protocols. Further a program developed on a single processor can be run unaltered on any number of processors since the allocation of tasks to processors is built in the run-time system of the host concurrent language. Ada and Occam [20,21,29] of INMOS were considered for the host language since both have message passing as their communication mechanism between concurrent program units and generic facilities for creating processes. However, Occam provides excellent concurrent primitives at the cost of good data structures and its primitive nature discourages the integrated system development at a higher level. Further Occam does not provide data types to handle dynamic situations while Ada's access types come in handy. Extensions to Ada are provided to facilitate a user in building a simulation model. The syntax of the extensions that provide the basic simulation primitives is given in Table 1.

Primitive	*Syntax of the extension*
Representation of an entity	**ENTITY** entity-name = list of attributes;
Representation of a process unit	**PROCESS** process-name; **Begin** .... end process-name;
Creation of an entity	**CREATE** entity-variable;
Flow of an entity	**SEND** entity-variable **TO** process-name; **RECEIVE** entity-variable;
Enqueing and dequeing an entity	**ENQUEUE** entity-variable; **DEQUEUE** entity-variable;
Removal of an entity form the system	**REMOVE** entity-variable;
Advance the clock of a process	**HOLD** time-unit;
Simulation termination condition	**STOP SIMULATION WHEN TIME** = time-unit;
Random varaite generators	**UNIFORM**(stream,parameters) **EXPONENTIAL**(stream,parameters) **POISSON**(stream,parameters) **NORMAL**(stream,parameters) **RANDOM**(stream)
Statistics collection	Automatic data collection on entities processes and queues in the simulation system & the following statements: **TALLY** real-variable; **ACCUMULATE** real-variable;

Table 1. Syntax of the Extensions that provide Simulation Primitives

The user model is processed by a preprocessor to replace the extensions by Ada statements and to create a simulation environment by instantiating a control module for each user defined process.

3.4 Three Different Ada Environments for Implementation

The primary aim of using Ada to build simulation environments has been to exploit and to study the utility of the package and generic concepts in generalizing the simulation tools and the tasking facilities in distributing the simulation by improving the concurrency [1,33]. The initial Ada implemenations at Texas A&M University involved the development of two systems that support process and event oriented simulation [13,32]. These two software systems were implemented and executed on a VAX 11/782 using the NYU Ada/Ed Translator/Interpreter version 1.1.4. The event oriented version was later modified to support distributed simulation by executing the support functions concurrently, on a VAX 11/750 [33,34]. Though the NYU Ada/Ed Translator is not a production compiler, the ease of generalizing the simulation concepts through the packages and generic units of Ada and the portability of Ada through various compilers encouraged us to test Ada in developing an integrated and concurrent simulation environment.

Recently Texas A&M University has acquired three more Ada compilers which overcome the very low productivity associated with the NYU Ada/Ed Translator significantly. The three compilers are Telesoft Ada and Digital Electronics Corporation Ada for the VAX 11/750 [9] and the ROLM Ada compiler [8] for the Data General MV/10000. With very few modifications the Ada programs written for one system have been easily run on the other systems. The strength of Ada thus lies in its portability and maintainability among

the different compilers and machines. Among the three systems Telesoft Ada has not been considered for distributed simulation application, since our current version does not support tasking.

3.5 Current Status

The protype of the above system is being implemented using the DEC Ada compiler running on VAX/VMS Version 4.1. The operation of the prototype will be analyzed by simulating the benchmark applications. Thus the outcome of this research will be a functional prototype of a discrete concurrent simulation system in which the hierarchical architecture is retained for the simulation support functions as parallel processes while user written portions of the model are simulated by the coexisting processes with message passing interfaces. Another advantage of utilizing a concurrent language as the host language is that the run-time system of the concurrent language will take care of assigning the concurrent units to the processors available. It will also provide a framework to analyze the sensitivity of the system to parameters like deadlock occurrences, processor utilization and total turnaround time.

4.0 SUMMARY

The language supported distributed simulation system is nearing its completion. The experience and insight gained from the design and the development of this system offers promise for exploiting the parallelism in the simulation language functions as a means for improving the performance of the system. This implementation approach also proves to be advantageous since it avoids the deadlock and synchronization problems and maintains the distributed implementation transparent to the user.

The second implemenation approach will retain the hierarchical distribution of simulation functions as in the first approach and will also provide concurrency features in its modeling of its user-written routines. Even though the second implementation approach has to deal with deadlock and synchronization problems and has to involve the user in the distribution of simulation model, it promises a better speed up from the distribution than the first approach. The future research at Texas A&M University will involve the complete implementation of the second approach and the performance evaluation of the distributed simulation systems implemented by both approaches.

REFERENCES

1. Bruno, G., "Using Ada for Discrete Event Simulation," *Software-Practice and Experience*, 14, 7, July 1984, pp. 685-695.

2. Bryant, R.E., "Simulation on a Distributed System," *Distributed Computing Conference*, 1979, pp. 544-552.

3. Chandrasekaren, U., Sheppard, S., "An Algorithm for Distributed Concurrent Simulation", (submitted for publication).

4. Chandy, K.M., Misra, J., "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, 24, 11, April 1981, pp. 198-206.

5. Comfort, J.C., "The Simulation of a Master-Slave Event Set Processor," *Simulation*, Volume 42, Issue 3, March 1984, pp. 117-124.

6. Comfort, J.C., Winquing, Y., and Li, Qiang., "The Design of a Multi-microprocessor Based Simulation Computer III," *Record of Proceedings of the 17th Annual Simulation Symposium*, March 1984, Tampa, Florida, pp. 227-241.

7. Concepcion, Arturo I., "Mapping Distributed Simulators onto Hierarchical Multibus Multiprocessor Architecture," *Distributed Simulation 1985, The 1985 SCS Multiconference*, San Diego, Ca, Vol. 15, No. 2, January 1985, pp. 8-13.

8. Data General Corp., "Ada Development Environment (ADE) (AOS/VS) User's Manual," *Data General Corporation*, April 1984.

9. DEC, "Developing Ada Programs on VAX/VMS ," *Digital Equipment Corporation*, Maynard, Massachusetts, February 1985.

10. Enslow Jr., P.H., "Multiprocessor Organization–A Survey," *Computing Surveys*, Vol. 9, No. 1, March 1977.

11. Fischer, Wayne., "The VMEbus Project," *Digest of Papers* Compcon Spring 84, pp. 376-378.

12. Fischer, Wayne., "IEEE P1014–A Standard for the High-Performance VME Bus," *IEEE Micro*, February 1985, pp. 31-41.

13. Friel, P., Sheppard, S., "Implications of the Ada Environment for Simulation Studies," *Proc. of the 1984 Winter Simulation Conference*, December 1984, pp. 477-489, .

14. Kiviat, P.J. "Simulation Languages," *On Computer Simulation Experiments with Models of Economic Systems*, T.H. Naylor, Ed., Wiley, New York, 1971, pp. 406-489.

15. Krishnamurthi, Murali., and Young, Robert E., "A Multitasking Implementation of System Simulation: The Emulation of an Asynchronous Parallel Processor Using a Single Processor," *Proceedings of the 1984 Winter Simulation Conference*, Dallas, November 1984, pp. 261-271.

16. Krishnamurthi, Murali., and Young, Robert E., "A Multitasking Implementation of System Simulation: The Emulation of an Asynchronous Parallel Processor for System Simulation Using a Single Processor," *Technical Report*, Volumes I and II, Department of Industrial Engineering, Texas A&M University, November 1984.

17. Krishnamurthi, Murali., and Young, Robert E., "Design of the Distributed Simulation System: Hardware Configurations," *Technical Report*, Department of Industrial Engineering, Texas A&M University, January 1985.

18. Krishnamurthi, Murali., and Lively, William M., "Interprocessor Communication Methods in Multiprocessor Systems," May 1985 (Submitted for publication).

19. Military Std., "Ada Programming Language," *Military Standard*, ANSI/MIL-STD-1983.

20. May, M.D., "Occam," *SIGPLAN Notices*, 18,4, April 1983, pp. 69-79.

21. May, M.D., Taylor, R.J.B., "Occam–an Overview," *Microprocessors and Microsystems*, 8, 6, Jul/Aug 1984.

22. Motorola Inc., "MVME110 VMEmodule Monoboard Microcomputer User's Manual," MVME110/D2, March 1983, Motorola Semi-conductor Products Inc., Phoenix, Arizona 85062.

23. Motorola Inc., "MVME900 Series Equipment User's Manual," MVME900/D1, October 1983, Motorola Semiconductor Products Inc., Phoenix, Arizona 85062.

24. Motorola Inc., "VME/10 Microcomputer System Reference Manual" M68KVSREF/D1, February 1984, Motorola Semiconductor Products Inc., Phoenix, Arizona 85062.

25. O'Grady, E.P., and Wang, C.H., "Multibus-based Parallel Processor for System Simulation," *Proceedings ofthe 1983 Simulation Conference*, Vancouver, B.C., Canada, July 1983, pp. 371-375.

26. Peacock, J.K., Wong, J.W., and Manning, E.G., "Distributed Simulation Using a Network of Processors," *Computer Networks*, 3, 1, Feb 1979, pp. 44-56.

27. Pimentel, J.R., "Real-time Simulation Using Multiple Micro-computers," *Simulation*, March 1983, pp. 93-104.

28. Pritsker, A., Alan, B., "The GASPIV Simulation Language," John Wiley & Sons, New York, NY 1974.

29. Product Review, "INMOS Launches Multiprocessor Language Occam," *A Product Review, Microprocessors and Microsystems* , 8, 1, Jan/Feb 1984, pp. 3-15.

30. Reynolds Jr, P. F., "Active Logical Processes and Distributed Simulation: An Analysis," *Proceedings of the 1983 Winter Simulation Conference*, pp. 263-264.

31. Sheppard, S., Philips, D.T., Young, R.E., "The Design and Implementation of a Microprocessor-based Distributed Digital Simulation System," *NSF Proposal RF-82-963*, 1982.

32. Sheppard, S., Friel, P., Reese, D., "Simulation in Ada: An Implementation of Two World Views," *Simulation in Strongly Typed Languages: Ada, Pascal, Simula*, 13, 2, February 1984, pp. 3-9.

33. Sheppard, S., Chandrasekaran, U., Murray, K., "Distributed Simulation Using Ada," *Distributed Simulation 1985, The 1985 SCS Multiconference*, San Diego, California, January 1985.

34. Sheppard, S., Young, R.E., Chandrasekaran, U., Krishnamurthi, M., Wyatt, D., "Three Mechanisms for Distributing Simulation," *Proc. of the 12th Conference of the NSF production Research and Technology Program*, Madison, Wisconsin, 1985.

35. VMEbus Manufacturers Group., "VMEbus Specifications Manual," Rev. B, August 1982.

36. Wyatt, Dana L., Sheppard, Sallie., and Young, Robert E., "An Experiment in Microprocessor-based Digital Simulation," *Proceedings of the 1983 Winter Simulation Conference*, December 1983, pp. 271-277.

37. Wyatt, Dana L., and Sheppard, Sallie., "A Language Directed Distributed Discrete Simulation System," *Proceedings of the 1984 Winter Simulation Conference*, Dallas, Texas, November 1984, pp. 463-464.

38. Young, Robert E., Sheppard, Sallie., and Krishnamurthi, M., "A Parallel Processor for System Simulation: The Design Rationale and Simulation Language Characteristics Suitable for Parallel Processing Applications," October 1984, (submitted for publication).