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.

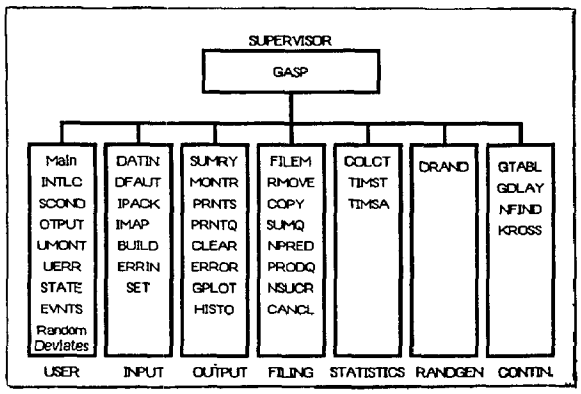

Figure 1. GASPIV Subprogram Paritition Groups

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.

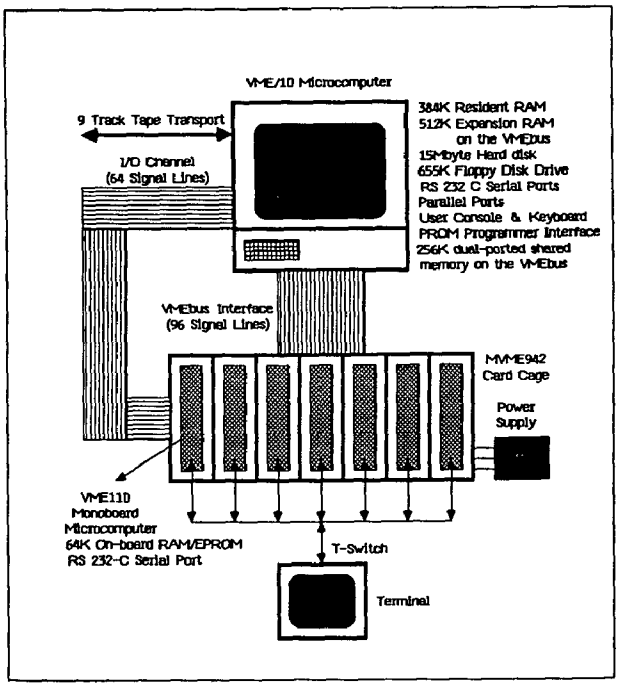

Figure 2.Hardware Setup of the Language Supported Distributed Simulation System

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.

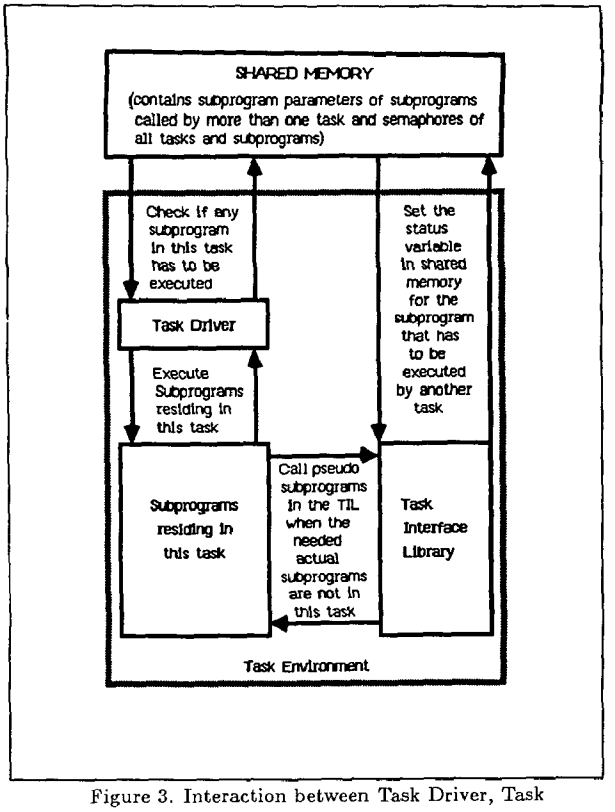

Figure 3. Interaction between Task Driver, Task Interface Library and the GASPIV Subprograms

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.

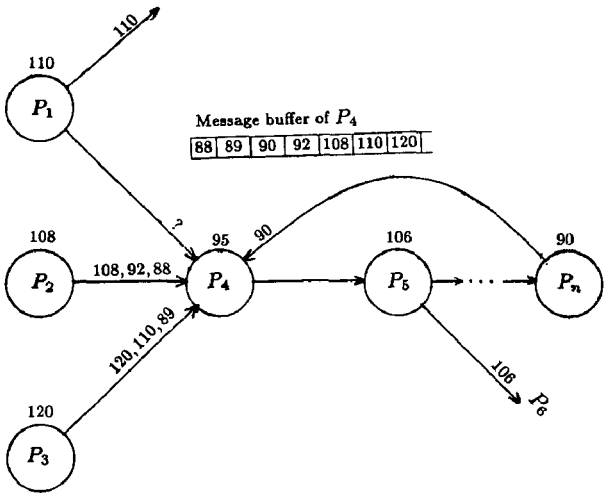

Figure 4. Blocking Situation

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

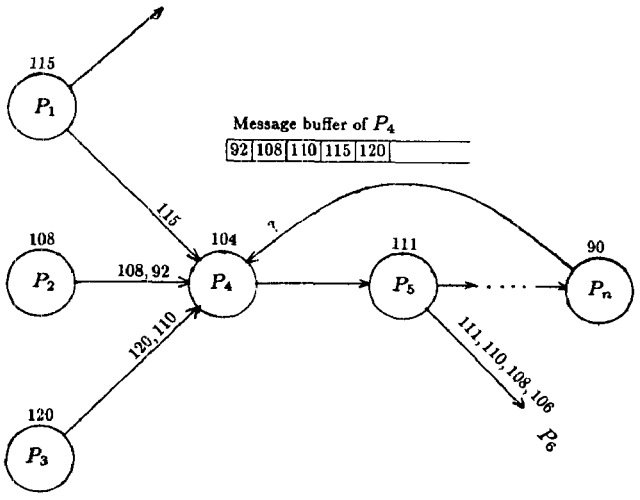

Figure 5. Deadlock Situation

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).

MURALI KRISHNAMURTHI

Murali Krishnamurthi is a doctoral student in the Department of Industrial Engineering and Manager of the Industrial Automation Laboratory at Texas A&M University. His research interests are automated manufacturing, simulation, and artificial intelligence and expert systems applications to manufacturing. He is currently working on an expert system tool evaluation project funded by the U. S. Air Force. He received his MS in Industrial and Systems Engineering from Ohio University and his BSME from the University of Madras, India. He is a member IIE, ORSA/TIMS, IEEE, SME and SCS.

Department of Industrial Engineering
Texas A&M University
College Station, Texas 77843
(409) 845-9363

USHA CHANDRASEKARAN

Usha Chandrasekaran is a doctoral student in the Department of Computer Science at Texas A&M University. Her research interests are simulation, high level languages, concurrency and parallel processing. She reveived her MS in Computer Science and BS in Electronics and Communication Engineering from the University of Madras, India.

Laboratory for Software Research
Texas A&M University
College Station, Texas 77843
(409) 845-4306

SALLIE SHEPPARD

Sallie Sheppard is an associate professor of Computer Science and is director of the Laboratory for Software Research at Texas A&M University. Her research interests include simulation, concurrent high level languagers, software engineering and expert systems. She has served as project director on a National Science Foundation project which has investigated the uses of multiple microprocessors working in parallel. She is chairman of the IEEE Computer Society Technical Committee on Simulation. In 1985 she received the Texas A&M University Former Student Faculty Achievement Award for Teaching.

Department of Computer Science
Texas A&M University
College Station, Texas 77843
(409) 845-5466