T-SE/13/6/14438

A Message-Based Approach to Discrete-Event Simulation

R. L. Bagrodia
K. M. Chandy
J. Misra

Reprinted from: IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Vol. SE-13, No. 6, JUNE 1987

A Message-Based Approach to Discrete-Event Simulation

RAJIVE L. BAGRODIA, MEMBER, IEEE, K. MANI CHANDY, SENIOR MEMBER, IEEE, AND JAYADEV MISRA, MEMBER, IEEE

Abstract—This paper develops a message-based approach to discrete-event simulation. Although message-based simulators have the same expressive power as traditional discrete-event simulation languages, they provide a more natural environment for simulating distributed systems. In message-based simulations, a physical system is modeled by a set of message-communicating processes. The *events* in the system are modeled by message-communications. The paper proposes the *entity* construct to represent a message-communicating process operating in simulated time. A general *wait until* construct is used for process scheduling and message-communication. Based on these two notions, the paper proposes a language fragment comprising a small set of primitives. The language fragment can be implemented in any general-purpose, sequential programming language to construct a message-based simulator. We give an example of a message-based simulation language, called MAY, developed by implementing the language fragment in Fortran. MAY is in the public domain and is available on request.

Index Terms—Discrete-event simulation, distributed system, entity, message, message-based simulation.

I. Introduction

THIS paper has the following goals:

1) Develop a message-based approach to discrete-event simulation.

2) Show that a class of simulation languages may be obtained by adding a small programming language fragment to general-purpose, sequential programming languages like Fortran, Pascal, C, etc.

3) Give an example of a message-based simulation language, called MAY, derived from Fortran. MAY runs on a variety of machines including VAX® 11/780 (under UNIX™), DEC 20 (under TOPS-20), IBM 3033 (under VM) and the IBM PC (under MS DOS).

A large number of discrete-event simulation languages including GASP [16], SIMSCRIPT [9], SIMULA [4], and GPSS [17] are currently available. Although message-passing can be simulated in these languages, none of them provide it as a basic language construct. Some recent research efforts have been directed towards designing simulation systems around message-based programming languages. Two efforts in this direction are the design of SAMOA [12], a discrete-event simulation package built upon Ada® [1], and a simulation system designed by Kaubisch and Hoare [8] built upon CSP [6]. This paper develops a class of message-based simulation languages by enhancing sequential programming languages with a small number of message-passing and process-representation primitives. We describe a small programming language *fragment* which can be implemented within the framework of any existing sequential language to construct a message-based simulator. The few primitives required for message-based simulations are constructs to:

1) *create* and *terminate* processes;
2) *send* messages to processes;
3) *wait* for messages and/or simulation time to elapse.

What is the advantage of simulations in which events are message-communications? Message-based simulations do not provide additional expressive power. Indeed there are systems for which the constructs of standard simulation languages such as GPSS are more natural than those of message-based simulation. However, for simulating distributed systems, message-based simulations appear to be more natural. A simulation program written in a traditional simulation language like GPSS is inherently a sequential program and is developed using the constructs of sequential programming. A message-based simulation, on the other hand, "looks" like a distributed program. It is natural to develop a message-based simulation of a message-based distributed system.

The rest of the paper is organized as follows. Section II presents an informal description of message-based simulation. Section III explains the philosophy and design details of the constructs proposed in this paper. Section IV describes the language fragment that needs to be added to general-purpose programming languages to obtain message-based simulation languages. Section V gives a description of MAY, a message-based simulation language derived from Fortran. Section VI presents a few example programs coded in MAY. Section VII discusses implementation issues. Section VIII is the conclusion.

II. Message-Based Simulation

This section presents an informal discussion of message-based simulation. In this paper we adopt the process view of simulation: a system being simulated is assumed to consist of a number of *physical processes* which inter-

Manuscript received May 31, 1984; revised January 31, 1985. This work was supported by a grant from the IBM Corporation.

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

IEEE Log Number 8714438.

®VAX is a registered trademark of Digital Equipment Corporation.

™UNIX is a trademark of AT&T Bell Laboratories.

®Ada is a registered trademark of the U.S. Department of Defense (Ada Joint Program Office).

act at discrete instants of time. The process interactions are called *events* and the time instants at which they occur, *event times*. Each physical process of the system is simulated by a *logical process*. As an example, consider the simulation of a doctor's office to determine the waiting time distribution of patients visiting the office. In the actual system, patients enter the office and wait in a queue to meet the receptionist. The receptionist fills out a medical form for the patient, who then waits to consult with the doctor. In the simulation, the doctor, receptionist, and patients are simulated by logical processes. Examples of events in this system are a patient entering a queue to talk to the receptionist and a patient completing his consultation with the doctor. Hereafter, we shall use terms like the *patient* process to refer to the logical process that simulates a physical patient process.

In message-based simulations, an event is the communication of a message. In the above example, the event of a patient entering the receptionist's queue may be simulated by a message being sent by the *patient* process to the *receptionist* process (and the associated queue); this message requests service from the *receptionist* process and is referred to as a message of type *request*, or simply as a *request* message. After sending the *request* message, the *patient* process waits to receive a *reply* message from the *receptionist* process. The *reply* message simulates the event of a patient having received his form from the receptionist. Internal actions of a physical process, i.e., actions that do not involve interactions with other processes in the system, are modeled either by the passage of simulation time or by the execution of sequential statements within the corresponding logical process.

We use a simulator clock to represent the passage of time in a simulation. The simulator clock advances in discrete steps, where each step simulates the passage of time between two events in the system. In message-based simulations, a process may wait until some time t specified by the process to receive a specific type(s) of message. While it is waiting for a specific message, other messages received by the process are enqueued in a message buffer associated with the process. The process may accept messages from its input buffer at a later point in the simulation, as explained subsequently. In our example, a patient entering the doctor's office may be willing to wait for up to 5 minutes to receive his form from the receptionist; if it takes any longer, the patient goes elsewhere. This can be simulated by the *patient* process waiting for up to 5 minutes of simulated time to receive the *reply* message from the *receptionist* process; if it takes any longer, the *patient* process simulates the departure of the patient from the system. A process waits for the passage of simulation time and/or to receive a specific type(s) of message by executing a wait statement, which has the form

wait t **for** b

where t represents an integer-valued expression and b a Boolean condition. The condition b is used to specify the messages that the process is ready to accept and t specifies the maximum time up to which the process is willing to wait for the message(s). The process ceases to wait either when the simulator clock reaches the value represented by t or if it is receives a message that satisfies the condition b, whichever occurs first. In our example, the receptionist may be at lunch, during which time he is willing to talk only to patients with an emergency; patients not having an emergency have to wait until the receptionist finishes his lunch. The event of a patient requesting emergency service from the receptionist is simulated by an *emergency* message being sent by the *patient* process to the *receptionist* process. The lunch-break of the receptionist may be simulated by executing the following **wait** statement:

wait t_1 **for (message-type** = *emergency*)

where t_1 represents the time at which the receptionist finishes his lunch-break; the keyword **message-type** is used by a process to refer to the type of message received by it. *Request* messages received by the *receptionist* process while it is willing to accept only *emergency* messages are enqueued in the message buffer associated with the *receptionist* process. If an *emergency* message is received by the *receptionist* process, it ceases to wait and executes the next statement in its code. If the simulator clock moves forward to t_1 and the *receptionist* process has not received a message of type *emergency*, then the process is said to have *timed-out*. The timing-out of a process is simulated by the process receiving a special message, called a *time-out* message, from the simulation monitor. Reception of a time-out message by a waiting process forces it to cease waiting.

On ceasing to wait, a process proceeds to the next statement in its code. For instance, on ceasing to wait, a process may execute a statement of the form

if **message-type** = *emergency* then do x
else if **message-type** = *time-out* then do y
    else do z.

The purpose of this informal discussion was to give the reader who is unfamiliar with message-based simulation a flavor of the technique.

III. Design Philosophy

This section contrasts the message-based approach to simulation with the approach adopted by some of the traditional discrete-event simulation languages.

Resources and *processes* are the two basic building-blocks of a simulation program. A *resource* is a passive object and may be represented by a simple variable or a complex data structure. A *process* on the other hand, is an independent, dynamic entity. *Processes* interact with each other and "use" the *resources* to achieve some objective. For instance, in the doctor's office example, the doctor and receptionist are resources that are used by the patient processes. A simulation program models the *dynamic* behavior of *processes*, *resources*, and their interactions as they evolve in time.

Most simulation languages treat *resources* and *processes* as two distinct concepts and provide separate primitives to model their behavior. Thus GPSS provides the built-in **storage** and **transaction** primitives, and SIMONE [7] the **monitor** and **process** primitives for this purpose. SIMULA uses the **class** concept to model *resources*, and a special subclass called the **process class** to model *processes*. However, as observed by Kaubisch and Hoare [8], the *process* and *resource* concepts are relative rather than absolute:

> "...the *processes* and *resources* of a simulation algorithm display a multilevel tree organization (with the *resources* at the bottom and intermediate levels, and the pure *processes* at the top level). Looking from the top down, every structure looks like a *resource*; looking from the bottom upward, they look like *processes*. Even the *processes* at the top level would look like *resources* if we were to add another level to the tree..."

The language fragment proposed in this paper uses a single program module, called an **entity,** to simulate both *processes* and *resources*. An entity is a sequential program module implemented in the host language with the following additional features: an entity may

1) create other entities;
2) terminate itself;
3) send messages to other entities; and
4) wait for the passage of time and/or receipt of messages.

The **entity** construct is a versatile modeling tool. It can be used to model a variety of different objects including abstract data types, recursive procedures, monitors, coroutines, processes, and resources. An entity-type models a class of objects. An instance of an entity-type represents a specific object of its class. Hereafter, we shall use the term entity to mean an instance of an entity-type. Each entity in the simulation has an independent data-space and its attributes cannot be accessed directly by other entities in the simulation.

Two primary operations are defined on an **entity:** *creation* and *termination*. The *create* primitive provided by our language fragment is semantically very similar to the **new** primitive provided in SIMULA. It allows multiple, independent instances of an entity-type to be created dynamically. An entity may also recursively create its own instances. Initially, every simulation consists of a single entity called **main.** Entity **main** does not model any real physical process; rather its purpose is to initiate and terminate the simulation.

Once created, an entity participates in the simulation until it ceases to exist. In SIMULA, if no references to an object exist, the object ceases to exist. Thus, a SIMULA object can be destroyed unconditionally by other objects in the simulation. The terminate mechanism proposed by our language fragment is different. The only mechanism by which an entity may terminate itself, is by executing the **end-entity** statement in its definition (see Section IV-C). An entity cannot be terminated by another entity in the system. On the basis of this design principle, even if no external references to an entity exist, the entity does not cease to exist. Attempts to refer to terminated entities result in error.

In a physical system, each process makes independent progress in time and many processes execute in parallel. In its simulation, the multiple processes of a physical system must be executed "simultaneously" on *one* processor. This simultaneity is achieved by interleaving the execution of different processes and executing them in a quasi-parallel fashion [5]. Quasi-parallel processing uses a time-measure that is distinct from real time. We refer to this as simulated time and the clock used to measure simulated time as the simulation clock. In a physical system, processes perform different tasks, each of which takes a certain amount of real time. In the simulation, this is represented by the process initiating the task and then being idle for the duration (as measured by the simulation clock) of the task completion. Scheduling primitives, provided by a simulation language, are used by a process to schedule its operations in simulated time. For instance, SIMULA provides a built-in procedure, **hold,** to allow a process to suspend its execution for a predetermined time duration. In our doctor's office example, we assume that the doctor takes 10 minutes to consult with a patient. In SIMULA, the **class** modeling a patient would contain the procedure call, **hold**(10) to represent this activity. The effect of executing the above statement would be to cause the process to wait for 10 units of simulation time to elapse. From the perspective of the patient process, if the value of the simulation clock was T before executing the procedure call, it would be $T+10$ after the call. However, the **hold** primitive is not sufficient to represent an activity that takes an unspecifiable amount of time. Thus, in the above example, when a patient enters the receptionist's queue, he cannot predict the amount of time he would have to wait before seeing the receptionist. To represent such activities, SIMULA provides the **passivate** statement, which when executed by a process causes it to enter an idle state; the process remains in the idle state until it is explicitly activated by another process.

The **wait-until** primitive is a general-purpose scheduling primitive which allows a process to wait until an arbitrary condition is satisfied. This primitive was first introduced in the language SOL [10], in the following form:

wait-until (*condition*)

where *condition* is a Boolean expression of arbitrary complexity. The **hold** primitive discussed earlier, can be expressed as a special form of the **wait-until** construct. For instance, **hold**(10) may be expressed as follows:

wait-until (*time* = *current-time* + 10)

where *time* refers to simulated time. In addition, this primitive can be used to simulate a variety of other situations. For instance, in the doctor's office example, the activity of a patient waiting in the receptionist's queue could be simulated by the patient process by executing the following statement

wait-until (*removed-from-queue*)

where *removed-from-queue* is a Boolean variable that is set to *true* by the receptionist process when the patient process has received the desired service.

The language fragment proposed in this paper uses a modified version of the **wait-until** construct, called the **wait** statement, which was introduced in Section II. In message-based simulation, an entity is in one of two states: *active*, if it is currently executing, or *waiting*, if it is not *active*. An *active* entity moves to the *wait* state by executing a **wait** statement. Every *waiting* entity has a **wait-time** and a Boolean predicate, called the **wait-condition**, associated with its *wait* state. The **wait-time** represents the *maximum* time that the entity can remain in the *wait* state. An entity in the *wait* state moves to the *active* state either when it receives a message that satisfies its **wait-condition** or if its **wait-time** has expired. We define a special message, called a *time-out* message, which is sent by the simulation monitor to a *waiting* entity, when its **wait-time** has expired.

The **wait** statement is used both as a scheduling primitive as well as a *receive* primitive for message-communications. Our communication protocol is based on buffered message-passing: execution of the *send* primitive causes a message to be deposited in a FIFO manner in the message buffer associated with the recipient entity. We now describe the **wait** statement in more detail: if an entity executes a **wait** statement at simulation time T (i.e., when the simulator clock time is T) and the wait statement is of the form

wait T' **for (message-type** $=$ M)

where T' represents a time value such that $T' >= T$, the meaning of the statement is as follows:

- The entity will cease waiting at time t, $T' >= t >= T$, if a message of type M is received by the entity at time t. In particular, if messages of type M are present in the message buffer of the entity at time T, the entity receives the first message of type M from the buffer and ceases to wait at T.
- The entity will cease waiting at time T', if no message of type M is received by it in the interval $[T, T')$. In this case, a *time-out* message is received by the entity at T'.

We note that if a message of type M is sent to the entity at time T', then the entity may first receive either the *time-out* message or the message of type M.

In message-based simulations, the communication of a message takes zero units of simulation time. For instance, in our doctor's office example, we assume that the time taken by a patient to walk from the receptionist's desk to the doctor's cabin is insignificant; thus, in the simulation, the transmission time for the message that models this event is zero. Nonzero transmission delays in physical systems can be modeled by causing the process sending (receiving) a message to wait for a certain time corresponding to the message-transmission time before (after) sending (receiving) the message. Alternatively, the communication medium may be modeled as a separate process which incorporates the transmission delay.

Many simulation languages, for instance GPSS, provide built-in language primitives for statistics collection, queue representation, and random number generation. Our language fragment treats these facilities as "options" rather than "standard equipment." As such, these facilities are provided to the programmer through a set of library entities and routines. An entity-type defined in the library, can be included in a user program by means of primitives provided by our language fragment. The library facility serves another useful purpose. A number of separate entities may be defined to model some primitive subsystems that comprise most distributed systems. Some examples of such subsystems are shared memory, ethernet, token-ring, processor, FIFO server, and priority server. Each of these subsystems may be programmed as an entity-type and stored in the library. The library can then serve as a tool-box to study the performance of various alternative configurations of a proposed distributed system.

IV. Constructs for Message-Based Simulation

In this section we present the message and process constructs provided by our language fragment. The programming language in which these constructs are to be embedded (e.g., Fortran, Pasacal, Algol, C, etc.) is called the host language. This discussion ignores anomalies that may arise in the implementation of these constructs in a specific host language. For instance, we have assumed that processes communicate exclusively through messages. However, in a Pascal implementation, processes may also share information by means of global variables; or in a Fortran implementation, through **common** blocks. We assume that specific implementations will handle such anomalies on an individual basis. Further, the discussion in this section ignores all syntactic issues. The next section, however, gives a description of the Fortran implementation of these constructs.

A. Entities

An entity-type models objects of a given type. The declaration of an entity-type is similar to that of a procedure or subroutine in the host language. An entity-type consists of an entity heading, a local variable declaration section, a message declaration section, and an entity body. The entity heading declares the name and formal parameters of the entity-type in a manner similar to the declaration of a procedure heading in the host language. An entity-type, however, is only allowed to have input parameters. The local variable section is identical to that of a procedure. The message declaration section is used to declare the various types of messages that may be received by all instances of this entity-type. A message declaration consists of the string **message** followed by a name and a parameter list. The structure of the parameter list is similar to that of a formal parameter list in the host language. The entity body consists of sequential statements of the host language (e.g., assignment statement, procedure call, etc.) with the following additional statements:

- **let** statement: used to create new entities.
- **end-entity** statement: used by an entity to terminate itself.
- **invoke** statement: used to send messages to other entities.
- **wait** statement: used to wait for the passage of simulation time or to wait to receive messages.

We now describe the semantics of each of the above statement types.

B. **Let** *Statement*

We define a scalar variable type called **entity-identifier.** Every entity in the simulation is assigned a unique identification number which is stored in a variable of type **entity-identifier.** Variables of this type are used exclusively to store the identifier of entities and no arithmetic operations can be performed on them. An entity is created by the execution of a **let** statement which has the following form:

let *e*1 **be** *entity-type-name*(actual parameter list)

where *e*1 is a scalar variable of type **entity-identifier.**

Execution of the above statement causes a new instance of the specified entity-type to be created; the identifier of the newly created entity is stored in *e*1. The formal parameters of the entity-type declaration are bound to the actual parameters in the **let** statement, as in the manner of a procedure call, at the point that the entity is created. The identifier *e*1 may be used to send messages to the entity. Every entity-type declaration contains a predefined local variable, called **myid,** which is of type **entity-identifier.** When a new entity is created, its identifier is automatically stored in its **myid.**

C. **End-entity** *Statement*

The **end-entity** statement is an executable statement, which when executed causes the entity to terminate. The statement has the form:

end-entity

An entity can only terminate by executing the **end-entity** statement in its entity definition. The entity is said to exist between the point that it is created to the point that it terminates itself. Attempts to refer to nonexistent entities result in error.

D. **Invoke** *Statement*

Messages are sent by one entity to another using an **invoke** statement which has the following form:

invoke *e*1 **with** *m*1(actual-parameter-list)

*e*1 must be of type **entity-identifier.** Execution of the above statement results in a message of type *m*1 being sent to the entity *e*1 provided:

- entity *e*1 exists.
- a message with name *m*1 has been declared in the message receive section of the entity-type declaration of entity *e*1.
- the types of the formal and actual parameters of message *m*1 match.

An attempt is made to deliver the message at the current value of the simulation clock. However, if the recipient entity *small-list* to represent a small list of integers. This message is stored in a buffer associated with the recipient entity and may be accepted by it subsequently (see discussion under **wait** statement).

E. **Wait** *Statement*

The **wait** statement is used by an entity to wait for the passage of simulation time and/or to receive messages. The statement has the following form:

wait *t* **for** *b*

where *t* is an integer-valued expression denoting time; and *b* is a Boolean condition of arbitrary complexity. This statement has been discussed extensively in the previous sections. In this section, we present a few examples of the more frequently used instances of this statement.

If the condition *b* is the Boolean constant *false*, the entity ceases to wait only when the simulation time reaches the time value specified in the wait statement. For instance, execution of the statement

wait *t* **for** *false*

will cause the entity to wait until the simulation time is *t*; all messages sent to it in the interim will be stored in its message buffer. In contrast, the condition *b* may be the Boolean constant *true*, as in the following statement:

wait *t* **for** *true*

If the message buffer of the entity is nonempty when the above statement is executed, the first message is removed from the buffer and delivered to the entity causing it to resume execution at the current value of the simulation clock. However, if the message buffer is empty, the entity will wait to receive a message. If no messages are received by the entity, and the simulation time reaches *t* units, a *time-out* message is sent to it at simulation time *t*, to force it to cease waiting.

The time part of the wait statement may be omitted. In this case, the **wait-time** associated with the entity is assumed to be infinity (represented by an arbitrarily large positive integer). In our doctor's office example, while the doctor is not busy, he sits in his office waiting for the next patient to arrive (we assume that the doctor never quits working!). The *doctor* entity would then contain a wait statement of the form

wait for (message-type = *request*)

F. Examples

This section illustrates how an entity may be used to model a variety of objects including abstract data types, monitors, coroutines, and recursive procedures. The section culminates in a complete example of a message-based simulation.

Abstract Data Types: An entity may be used to represent an abstract data type. Operations on the data type are performed by means of messages sent to the entity modeling the data type. As an example, we describe an entity *small-list* to represent a small list of integers. This example has been adapted from [6]. Two operations may be performed on this list: *insert* to insert an integer in the list; and *belongs* to check if a given integer belongs to the list. The insert operation may be performed by sending an *insert* message to the entity; to perform the *belongs* operation the user process sends a *belongs* message to the entity; the entity responds with a *position* message giving the position of the element in the list (it returns 0, if the element does not belong to the list).

```
entity small-list (list-size : integer);

{ Local Variable Declaration Section }
list-array:array[ 1 . . list-size] of integer;
pos,size : integer;

{ Message Receive Declaration Section }
   message insert(element : integer);
   message belongs(element : integer; sender-id : entity-identifier);

{ Entity Body }
   size := 0;

   while true do
   begin
{ wait indefinitely to receive the next message   }
      wait for true;

{ the procedure search is used to locate the position,
pos of the element in the array; pos is set to 0 if
the element is not present.   }
      search(list-array,size,element,pos);

      if (message-type = insert) and (pos = 0) then
      begin
         if (size = list-size) then overflow-error
         else begin
            size := size + 1;
            list-array[size] := element;
         end;
      end
      else if (message-type = belongs) then
         invoke sender-id with position(pos);
   end;
end-entity;
```

Recursive Procedures: To illustrate how an entity may be used to model recursive procedures, we define an entity *sieve* which recursively sieves out all prime numbers from a sequence of consecutive natural numbers beginning with the smallest prime, 2. The entity *sieve* is defined with one parameter. Multiple instances of the *sieve* object are created recursively as needed. For instance, the *i*th *sieve*, say s_i, is created with the *i*th prime number, say p_i as its parameter, by the (i-1)th *sieve*. Subsequently, s_i sieves out all multiples of p_i from the sequence of numbers passed to it by sieve s_{i-1}. The next prime number in the sequence, say p_{i+1}, is the smallest integer greater than p_i, which was not seived out by sieves $s_1 \, . \, . \, s_{i-1}$. When s_i receives this number, it creates the next sieve object, s_{i+1} with p_{i+1} as its parameter.

A driver entity is used to initiate the program. This entity creates the first *sieve* entity s_1 with the smallest prime, 2 as the actual parameter. It then passes consecutive natural numbers to s_1 via a stream of *send* messages. The code for the driver routine has been omitted for brevity.

```
entity sieve(prime: integer);

{ Local Variable Declaration Section }
next-sieve : entity-identifier;

{ Message Receive Declaration Section }
message send(number:integer);

{ Entity Body }
{ Wait to receive the next integer in the sequence }
      wait for ( message-type = send);
{     The first message received contains the next prime
      in the sequence Create the next sieve process,
      using number as the parameter }
   let next-sieve be sieve(number);

   while true do
   begin
      wait for (message-type = send);

      {     From the subsequent messages received,
            sieve out all multiples of prime and pass the
            rest to sieve next-sieve.      }
      if (mod(number,prime) < > 0) then
            invoke next-sieve with send(number);
   end;
end-entity;
```

Coroutines: Coroutines can be naturally modeled by entities. As an example, consider an unbuffered producer–consumer system. The producer process produces a value and sends it to the consumer process. In the absence of a buffer, the producer process must wait until the value has been accepted by the consumer process, before it can produce the next value. The two coroutine-like processes may be modeled by the *producer* and *consumer* entity-types, respectively.

```
entity producer;

{ Local Variable Declaration Section }
   value : integer;

{ Message Receiver Declaration Section }
   message more(consumer-id: entity-identifier);
```

```
{ Entity Body }
  while true do
  begin
    produce a value;
    wait for (message-type = more);
    invoke consumer-id with data(value);
  end;
end-entity;

entity consumer(producer-id: entity-identifier);

{ Message Receiver Declaration Section }
  message data(value:integer);

{ entity body }
  while true do
  begin
    invoke producer-id with more(myid);
    wait for (message-type = data);
    consume the value;
  end;
end-entity;
```

Monitors: This example illustrates how a monitor may be modeled by an entity. We use the example of a bounded buffer. The buffer is used to smooth out the variations in speed of output by a producer process and input by a consumer process. The buffer is modeled by a *buffer* entity. The entity receives *send* messages from the *producer* entity and *more* messages from the *consumer* entity. If the *buffer* receives *more*(*send*) messages when it is empty (full), the messages are stored in a queue associated with the *buffer* and accepted by it only when it is nonempty (nonfull). We assume that the *consumer* entity contains the definition of a message *receive*, to receive data from the *buffer*. Further, it is assumed that when requesting data, the *consumer* entity sends its identifier as a message parameter to the *buffer* entity. This is required by the *buffer* to send messages to the *consumer* entity.

```
entity buffer(buffer-length:integer);

{ Local Variable Declaration Section }
  in, out : integer;
  store:array[1 . . buffer-length] of integer;

{ Message Receive Declaration Section }
  message send(value:integer);
  message more(consumer: entity-identifier);
{ Entity Body }
{  initialize the buffer}
  in:=0
  out:=0
  while true do
  begin
    if (in = out + buffer-length) then
      wait for (message-type = more)
    else if (in = out) then
          wait for (message-type = send)
        else wait for true;
    if (message-type = send) then
    begin
      store((mod(in,buffer-length)) + 1) := value;
      in := in + 1;
    end
    else if (message-type = more) then
      begin
        value:=store((mod(out, buffer-length)) + 1);
        out:=out+1;
        invoke consumer with receive(value);
      end;
  end;
end-entity;
```

Simulation Model of a Doctor's Office: We develop a message-based simulation model of the doctor's office described earlier. This is a complete simulation example which illustrates how processes may be modeled by entities. To summarize, in the physical system, patients enter the office through a door, meet the receptionist to obtain their forms, consult with the doctor and then exit from the office. The door may be modeled by a *source* entity which creates *patient* entities at a rate equal to the arrival rate of patients at the door. We represent the interarrival time between two patients by the variable *next-arrival*. The identifiers of the *doctor* and *receptionist* entities are passed to the *patient* entities as entity parameters. On being created, a *patient* entity requests service from the *receptionist* entity by sending it a *request* message, and waits to receive the *reply* message. On receiving the *reply* message from the *receptionist*, it requests service from the *doctor* entity by sending it a *request* message. It then waits indefinitely for the *reply* message which indicates the completion of the examination. On receiving this message, the *patient* entity terminates itself.

We model both the *doctor* and *receptionist* entities as instances of one entity-type, called *server*. The *server* entities are assumed to exist forever. When idle, they accept a *request* message sent to them, wait for a certain time corresponding to the service time of the request to elapse, and then wait for the next *request*. *Request* messages received by a *server* entity, while it is busy serving another *request*, are buffered; after the *server* entity finishes serving the current *request*, it accepts the first *request* message from the buffer.

The entity-type *main* initiates the simulation by creating the *doctor*, *receptionist* and *source* entities. The simulation is terminated when all entities have been terminated or when the value of the simulator clock exceeds the value of a keyword **max-simulation-time.** The keyword **clock** represents the current value of the simulator clock. The program representing the above simulation model is presented in Pascal-like pseudo-code in Fig. 1.

The next section presents a detailed description of a message-based language called MAY in which Fortran is the host language. This section may be skipped by those who want to have an understanding of how message-based languages may be developed from host languages but have less interest in details of a Fortran implementation.

```
entity main;

{ Local Variable Declaration Section }
   door, doctor, receptionist:entity-identifier;
   form-fill-time, consult-time, next-arrival, sim-period:integer;

{ Message Receive Declaration Section }
{ Message type time-out is implicitly defined for every entity  and
   must not be defined by the user.  Since no other message types are
   used by this entity, this section is empty }
   message ;

{ Entity Body }
   read (form-fill-time, consult-time, next-arrival,sim-period);
   let receptionist be server(form-fill-time);
   let doctor be server(consult-time);

   let door be source(next-arrival, receptionist, doctor);
   max-simulation-time := sim-period;
   wait (clock+ sim-period) for false;
end-entity;


entity  source(inter-arrival : integer; server1,server2 : entity-identifier);

{ Local Variable Declaration Section }
   next-patient:entity-identifier;

{ Message Receive Declaration Section }
   message;

{ Entity Body }
   while true do
   begin
      let next-patient be patient(server1,server2);
      wait (clock + inter-arrival) for false;
   end;
end-entity;


entity   patient(reception, doc:entity-identifier);

{ Message Receive Declaration Section }
   message reply;

{ Entity Body }
   invoke reception with request(myid);
   wait for (message-type = reply);
   invoke doc with request(myid);
   wait for (message-type = reply);
end-entity;


entity  server(mean-service-time : integer);

{ Message Receive Declaration Section }
   message request(patient-id : entity-identifier);

{ Entity Body }
   while true do
   begin
      wait for (message-type = request);
      wait (clock+mean-service-time) for false;
      invoke patient-id with reply;
   end;
end-entity;
```

Fig. 1. Message-based simulation model of a doctor's office.

V. Description of MAY

This section gives a concise definition of MAY, a message-based simulation language implemented in Fortran. A complete description of MAY is given in [13]. The syntax of MAY statements is given in BNF using the following meta-symbols:

|	Represents alternatives; for instance $a\|b$ implies the presence of symbol *a or* symbol *b* in the corresponding statement.
{ }	Symbol(s) occurring within braces are optional in the corresponding statement.
identifier	Any Fortran identifier of type integer.

A. Entity Definition

A MAY entity has the following structure:

⟨entity-definition⟩ ::= ⟨entity-heading⟩ {⟨variable-declaration⟩} ⟨message-declaration⟩ ⟨entity-body⟩

Entity Heading: The entity heading contains the name of the entity-type and specifies the parameters that may be needed for its definition. The parameters may be scalar integer variables or one-dimensional array of integer variables. A scalar variable may optionally be followed by a size specification, which specifies the maximum permissible value for the corresponding actual parameter. An array parameter may be dimensioned by a scalar parameter whose maximum size has been specified. The entity heading has the following syntax:

⟨entity-heading⟩ ::= **entity** ⟨entity-name⟩ {(⟨parameter-list⟩)}
⟨entity-name⟩ ::= **identifier**
⟨parameter-list⟩ ::= ⟨parameter⟩ {,⟨parameter-list⟩}
⟨parameter⟩ ::= ⟨simple-par⟩ | ⟨array-par⟩
⟨simple-par⟩ ::= **identifier** | **identifier : positive integer**
⟨array-par⟩ ::= **identifier[identifier** | **positive integer]**

As an example, consider the heading for an entity-type *histo*, representing a histogram used to measure the frequency with which the value of a variable occurs within different specified intervals. The parameters required to represent a general histogram object may be:

ntrval: Number of intervals for the frequency distribution; we choose a maximum of 50 intervals.
minval: The minimum expected value for the variable.
ntrarr[i]: the upper bound for the *i*th interval; $i = 1 \cdot\cdot ntrval$.

The heading would then be coded as:

entity *histo*(*ntrval*: 50, *minval*, *ntrarr*[*ntrval*])

Since the array *ntrarr* is dimensioned by *ntrval*, the size of the actual array parameter cannot exceed 50. Parameters are treated as constants in the body.

Variable Declaration Section: The variable declarations of an entity-type consist of Fortran declaration statements and MAY **local integer** statements. The values of variables declared as local integers are defined even when the entity is waiting (i.e., not executing). MAY local variables may be integer scalar variables or one-dimensional arrays of integer variables. The **local integer** statement has the following syntax:

⟨var-declaration⟩ ::= **local integer** ⟨var-list⟩
⟨var-list⟩ ::= ⟨variable⟩ {, ⟨var-list⟩}
⟨variable⟩ ::= ⟨simple-var⟩ | ⟨array-var⟩
⟨simple-var⟩ ::= **identifier**
⟨array-var⟩ ::= **identifier (identifier** |**positive integer)**

The *histo* entity-type discussed above, may use the following local integers;

values: An array which stores the frequency of occurrence of the values of a variable in the different intervals.
totval: Stores the total of all values.
novals: Stores the total number of values in all intervals.

These are declared by the following MAY statement:

local integer *values* (*ntrval*), *totval, novals*

Message Declaration Section: This section declares the types of messages that may be received by entities of a given type. Each message declaration has the following syntax:

⟨message-statement⟩ ::= **message** [message-name⟩ {(⟨parameter-list⟩)}
⟨message-name⟩ ::= **identifier**
⟨parameter-list⟩ ::= ⟨parameter⟩{,⟨parameter-list⟩}
⟨parameter⟩ ::= ⟨simple-par⟩ | ⟨array-par⟩
⟨simple-par⟩ ::= **identifier** | **subscripted identifier**
⟨array-par⟩ ::= **identifier [positive integer]**

Messages contain only input parameters which may be scalars or variable length one-dimensional array variables. All message parameters must be of type integer. Two parameterless message types **init** and **tmout** are defined by the translator for every entity and must not be defined by the user.

init: As soon as it is created, an entity is invoked with the **init** message. The purpose of this message is to cause execution of the initializing statements in the entity description.
tmout: An entity is invoked with a **tmout** message when the simulation time reaches the value specified by the last **wait** statement that was executed by the entity.

Consider the declaration of a message to be used to pass a set of values to an entity of type *histo*.

message *insert*(*novals*, *values*[20])

The above declaration defines a message called *insert* which has two parameters, the second parameter being an array of size 20. The corresponding actual parameter (in the **invoke** statement) is expected to be an array. The array may be of any size, up to a maximum of 20.

Entity Body: The entity body consists of a set of executable Fortran and MAY statements which may include the **let, invoke,** and **wait** statements. The last statement in the body of the entity definition must be the end-entity statement, which when executed causes the termination of the entity-instance (see discussion on entity termination below).

B. Entity Creation

An entity can create an instance of an entity-type by executing a **let** statement which has the following syntax:

⟨let-statement⟩ ::= **let** ⟨entity-instance-name⟩ **be** ⟨entity-name⟩ { (⟨actual-par-list⟩) }
⟨entity-instance-name⟩ ::= **identifier** | **subscripted identifier**
⟨entity-name⟩ ::= **identifier**
⟨actual-par-list⟩ ::= ⟨parameter⟩ {, ⟨actual-par-list⟩ }
⟨parameter⟩ ::= **identifier** | **subscripted identifier** | **array name** |**integer**

The identifier of the created entity is stored in the variable represented by ⟨entity-instance-name⟩. The type **entity-identifier** is implemented in MAY as an integer type.

Consider the creation of an instance of entity *histo* with the following attributes:

Number of intervals = *num*.
Minimum value = *min*.
Upper bound for the *i*th interval(*i* : 1 . . 20) = *bounds*(*i*) where *bounds* is an array variable whose dimension is 20.

An instance of the *histo* entity with the above attribute values is created by the following **let** statement:

let *hist*1 **be** *histo*(*num*,*min*,*bounds*)

where *hist*1 has been declared as a local variable of type integer in the creator module. The identifier of this entity is stored in variable *hist*1.

C. Entity Termination

An entity terminates itself by executing an end-entity statement which has the following syntax:

⟨end-entity-statement⟩ ::= **ende**

D. Invoke Statement

A message may be sent to an entity by means of an **invoke** statement which has the following syntax:

⟨invoke-statement⟩ ::= **invoke**⟨entity-instance-name⟩ **with** ⟨message-name⟩ {(⟨parameter-list⟩)}
⟨entity-instance-name⟩ ::= **identifier** | **subscripted indentifier**
⟨message-name⟩ ::= **identifier**
⟨parameter-list⟩ ::= ⟨parameter⟩{,⟨parameter-list⟩}
⟨parameter⟩ ::= ⟨simple-par⟩ | ⟨array-par⟩
⟨simple-par⟩ ::= **identifier** | **integer** | **subscripted identifier**
⟨array-par⟩ ::= **identifier[identifier** |**positive integer]**

A message of type *insert* may be passed to the entity *hist*1 (if it exists), by executing the following statement:

invoke *hist*1 **with** *insert*(5,*values*[5])

The message has two parameters, the second being an array of 5 elements.

E. Wait *Statement*

The **wait** statement has the following syntax:

```
<wait-statement> ::= wait { <t> } { for <b> }
<t>              ::= FORTRAN integer arithmetic
                        expression
<b>              ::= FORTRAN boolean condition
```

If the **for** part of the wait statement is omitted, the condition ⟨b⟩ is assumed to be the Boolean constant *true*. If the time-part of the **wait** statement is omitted, the variable ⟨t⟩ is assumed to the **maxint**, an arbitrarily large number.

F. Append *Statement*

The entity-types defined in the MAY library can be included in MAY programs by means of an **append** statement. A library entity may be used as any of the other entities defined in the program. An **append** statement has the following syntax:

```
<append-statement> ::= append <entity-name-list>
<entity-name-list> ::= <entity-name>
                          {, <entity-name-list>}
<entity-name>      ::= identifier
```

VI. MAY Examples

We now present a few modeling and simulation examples coded in MAY.

A. *Representation of Processor Configurations*

A variety of processor configurations can be represented in MAY. We present a few examples illustrating how processors interconnected in a pipeline, organized as a hierarchy, or connected in a two-dimensional mesh can be modeled in MAY.

A pipeline of n processors may be created by executing the following code segment:

```
   do 10 i = 1, n
      let pipe(i) be processor
10 continue
```

However, processor *pipe*(i) cannot communicate with *pipe*(i + 1) unless it knows its identity. There are several ways in which a processor may obtain the identity of its neighbor: after all processors have been created, a message may be sent to each processor giving it the identity of its neighbor; alternatively, every processor (except the last one in the pipeline) may be given the identity of the next processor in the pipeline as an entity parameter. This is achieved by executing the code segment shown below:

```
   let pipe(n) be processor(0)
   do 10 i = n - 1, 1, -1
      let pipe(i) be processor(pipe(i + 1))
10 continue
```

A binary tree of $2n + 1$ processors, where n is a positive integer, may be created by executing the following statements. The processors form a complete binary tree in that all leaf nodes occur at the same level in the tree.

```
   let node(1) be processor(0)
   do 10 i = 1 , n
      let node(2i) be processor( node(i))
      let node(2i+1) be processor( node(i))
10 continue
```

In the above example, each node knows the identity of its parent. To permit a node to communicate with its child nodes, each node (except the root node) may send its identity to its parent node in a message.

As another example, consider a mesh of $n^2 + 2n$ processors, in which each of the processors represented by processor(i, j); $i: 1 \cdots n$, $j: 1 \cdots n$, communicates with processor($i + 1, j$) and processor($i, j + 1$). Processors(i, $n + 1$); $i: 1 \cdots n$, and processors($n + 1, j$); $j: 1 \cdots n$, are sink processors. This configuration may be used to implement a fast parallel algorithm to compute the product of two matrices [6]. The above configuration may be created by the following code segment:

```
C     create the sink processors.
   do 10 i = 1 , n
         let mesh(i, n+1) be processor(0, 0)
         let mesh(n+1, i) be processor(0, 0)
10    continue
   do 20 i = n,1,-1
   do 20 j = n,1,-1
         let mesh(i, j) be processor(mesh(i,j+1),
            mesh(i+1,j))
20    continue
```

B. *Simulation of a Doctor's Office*

This program illustrates the simulation of the doctor's office example introduced earlier in the paper. The program is a refinement of the pseudo-code developed in Section IV-F. The doctor's office is modeled as a simple queuing network with a *source* entity simulating the door and two *server* entities which simulate the receptionist and doctor, respectively. In addition to the entities discussed earlier, the program uses a MAY library entity-type called *histo*. An instance of the *histo* entity-type is used in the program to generate a frequency distribution of the total time spent by patients in the system.

A MAY program consists of a collection of user-defined entities, library entities and Fortran subroutines, with no "main program" segment. However, every MAY program must define a parameterless entity called **main.** This entity is used to initiate the MAY simulation. The simulation is normally terminated when all entities have

```
      entity main

      local integer  doctor, recept,door,hist1,output
      integer minval,ntrarr(10),ntrval

      message

C Create entities recept and doctor with mean service times of 2500 and 5000.
      let recept be server(2500)
      let doctor be server(5000)

C   Create an instance of the histo entity, which has three parameters
C   minval : Represents the minimum expected value of the measure.
C   ntrval : Represents the no. of intervals desired in the frequency dist.
C   ntrarr(i) : Upper bound of the ith interval; i: 1..ntrval

      output=6
      minval=0
      ntrval=10
      do 10 i = 1,ntrval
10         ntrarr(i) = 1000 * i
      let hist1 be histo(ntrval,minval,ntrarr)

C   Create an instance of the source entity with inter-arrival time = 10000
      let door be source(10000,recept,doctor,hist1)

C    The simulation is to be carried out for 500000 time units.
      izstim=500000
      wait clock+izstim

C    Terminate the various entities created by main.
      invoke door with dump(output)
      invoke recept with trmnt
      invoke doctor with trmnt
      invoke hist1 with dump(output)
      ende

C*************************************************************************
C*************************************************************************

      entity  source(arrvl,servr1,servr2,hist1)

      local integer nopats
      real expon
      integer  time,output

      message dump(output)

      nopats =0
10  continue
C    Create an instance of the patient entity.
      nopats = nopats+1
      let p1 be patient(servr1,servr2,hist1)
C    The function expon returns a random value exponentially distributed
C    about the mean value.
      time = int(expon(arrvl))
      wait clock + time
      if (msg .ne. dump)go to 10

C    ASSERT: Message is a dump message
      write(output,*)'No. of patient processes created',nopats
      ende

C*************************************************************************
C*************************************************************************

      entity  server(mtime)
C   mtime - mean service time for the server.

      local integer srvid
      integer time,hisid
      real     expon

      message requst(hisid)
      message trmnt

C   wait to receive the next message.
10    continue
      wait maxint

      if (msg .ne. requst)go to 60
C   Serve incoming requests.
      srvid=hisid
      time=int(expon(mtime))
      wait clock+time for (msg .eq. trmnt)
C   No request messages are accepted while the server is busy.

C   A tmout or a trmnt message was received.
      if (msg .eq. tmout) then
            invoke srvid with reply
C   previously buffered messages(if any), can now be accepted by the server.
            go to 10
      endif
60  ende

C*************************************************************************
C*************************************************************************

      entity patient(recept,doctor,hist1)

      local integer  start
      real  expon,ran
      integer time

      message  reply

      start = clock
      invoke recept with requst(myid)
      wait  for (msg.eq.reply)

      invoke doctor with requst(myid)
      wait  for (msg.eq.reply)
      invoke hist1 with insert(clock - start)

      ende

C*************************************************************************
C*************************************************************************

C    The definition of entity-type histo is stored in the MAY library.
C    Two message-types are defined for this entity:
C    insert(value) : Used to receive an integer value.
C    dump(output)  : Used to print a frequency distribution of the
C                          values on the file specified by output.

      append histo
```

Fig. 2. MAY program to simulate a doctor's office.

been terminated. It is terminated before this point if the value of the simulator clock, represented by **clock** is greater than the value of a MAY keyword **izstim.** The variable **izstim** may be assigned an integer value within any entity in the MAY program. The keyword **maxint** represents an arbitrarily large integer. The keyword **msg** is used by an entity to refer to the type of the last message received by it. The complete code for the program is given in Fig. 2.

VII. Implementation Issues

Message-based simulation languages may be developed by implementing the language fragment described in this paper in any sequential programming language. Two simulation languages, MAY [3] based on Fortran, and SOF [15] based on Pascal, have been implemented at the University of Texas at Austin. A preprocessor is used to translate programs written in the simulation language (e.g., MAY) into the corresponding host language (e.g., Fortran). The output from the preprocessor is compiled using a standard host language compiler (e.g., f77 under UNIX). A specific preprocessor can be implemented very rapidly. The MAY preprocessor, which was written in Fortran, took less than two man-months to implement. Since the MAY statements constitute a small percentage of the simulation program, the overhead associated with the translation is low. The preprocessor contains extensive error detection facilities. In particular, syntactic checks are provided to ensure that all message-types and entity-types referred to in the program have been defined. In addition, code is generated to check for run-time errors like references to terminated entities, exceeding the specified maximum size of a formal entity parameter, etc.

The implementation provides a trace facility which can be used to trace the various messages received by an entity and/or print the state of an entity at various points in the simulation. Using this facility, it is possible to dynamically trace the execution of specific entities or all entities of a given entity. The tracing may be performed at different levels depending to the detail to which the state information of an entity is desired. Further, an entity may be traced over specific time period, or tracing initiated on a certain condition corresponding to a possible error in the program.

VIII. Conclusion

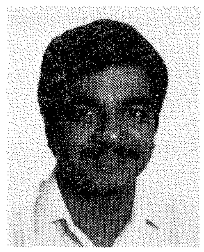
This paper developed a message-based approach to discrete-event simulation. Message-based simulators are a natural way to model distributed systems. The paper described how a class of message-based simulation languages could be developed by enhancing general-purpose sequential languages with a few message-passing and process representation constructs. The features needed for message-based simulation were illustrated by means of a Fortran derived message-based simulation language, called MAY. The language developed is simple, easy to learn and portable. It is currently being used to study the performance of distributed simulation techniques [11], performance of various distributed algorithms [2], and for the modeling and simulation of certain distributed systems [14].

Acknowledgment

We are thankful to two anonymous referees for their comments on an earlier version of this paper, which resulted in substantial improvements in the presentation of the material. We are also grateful to F. May of IBM, Austin, for his help and encouragement for this research.

References

[1] *Reference Manual for the Ada Programming Language*, U.S. Dep. Defense, 1982.
[2] R. Bagrodia, "An environment for developing distributed applications," Ph.D. dissertation, Dep. Comput. Sci., Univ. Texas at Austin, in preparation.
[3] ——, "May: A process based simulation language," Master's thesis, Dep. Comput. Sci., Univ. Texas at Austin, 1983.
[4] O. J. Dahl, B. Myhrhaug, and K. Nygaard, *Simula 67 Common Base Language*. Oslo: Norwegian Computing Centre, 1970.
[5] W. R. Franta, *The Process View Of Simulation*. New York: Elsevier North-Holland, 1977.
[6] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978.
[7] W. H. Kaubisch, R. H. Perrot, and C. A. R. Hoare, "Quasiparallel programming," *Software—Practice and Experience* vol. 6, no. 3, pp. 341-356, July-Sept. 1976.
[8] W. H. Kaubisch and C. A. R. Hoare, "Discrete event simulation based on communicating sequential processes," Dep. Comput. Sci., The Queen's Univ., Belfast, N. Ireland, Tech. Rep., 1983.
[9] P. Kiviat, R. Villareau, and H. Markowitz, *The SIMSCRIPT II Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, 1969.
[10] D. Knuth and J. McNelay, "SOL—A symbolic language for general purpose systems simulation," *IEEE Trans. Electron.*, pp. 401–408, Aug. 1964.
[11] D. Kumar, "Distributed simulation," Ph.D. dissertation, Dep. Comput. Sci., Univ. Texas at Austin, in preparation.
[12] G. Lomow and B. Unger, "Process view of simulation of Ada," in *1982 Winter Simulation Conf.*, 1982, pp. 77–86.
[13] *MAY Reference Manual*, Dep. Comput. Sci., Univ. Texas at Austin, 1983.
[14] C. Morelos, "Performance analysis of local area networks," Master's thesis, Dep. Comput. Sci., Univ. Texas at Austin, 1985.
[15] J. Palmer, "SOF: A distributed systems simulation language," Master's thesis, Dep. Comput. Sci., Univ. Texas at Austin, 1983.
[16] A. Pritsker, *The GASP IV Simulation Language*. New York: Wiley, 1974.
[17] T. Schriber, *Simulation Using GPSS*. New York: Wiley, 1974.



Rajive L. Bagrodia (S'79–M'81) received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Bombay, India, in 1981 and the M.A. and Ph.D. degrees in computer science from the University of Texas at Austin in 1983 and 1987, respectively.

He will be joining the Department of Computer Science at the University of California, Los Angeles, as an Assistant Professor in July 1987. He has worked as a consultant for the Software Technology Program at Microelectronics & Computer Technology Corp., Austin. His current areas of research include performance evaluation, concurrent programming, and software engineering.

K. Mani Chandy (SM'84) received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Madras, India, the M.S. degree in electrical engineering from the Polytechnic Institute of Brooklyn, Brooklyn, NY, and the Ph.D. degree in operations research from the Massachusetts Institute of Technology, Cambridge, in 1965, 1966, and 1969, respectively.

From 1966 to 1967 he worked for Honeywell EDP and from 1969 to 1970 he worked at the IBM Cambridge Scientific Center. He has also been a consultant to the Computer Sciences Department, Thomas J. Watson Research Center. He is presently Professor of Computer Sciences and Electrical Engineering, University of Texas, Austin. His current research interests include concurrent programming and computer performance.

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

Jayadev Misra (M'86) recevied the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Kanpur, in 1969, and the Ph.D. degree in computer science from The Johns Hopkins University, Baltimore, MD, in 1972.

He worked for IBM, Federal System Division, from January 1973 to August 1974. Since 1974 he has been with the Department of Computer Sciences, University of Texas, Austin, where he is currently a Professor. He spent 1983–1984 as a Visiting Professor at the Computer Systems Lab, Stanford University, Stanford, CA.

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