THE IMPLEMENTATION OF FOUR CONCEPTUAL FRAMEWORKS FOR SIMULATION MODELING IN HIGH-LEVEL LANGUAGES

OSMAN BALCI

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061-0106

1 August 1988

Published in *Proceedings of the 1988 Winter Simulation Conference*, pp. 287-295.

ABSTRACT

This is a tutorial paper on how to implement a simulation model in a high-level programming language (e.g., C, Pascal, FORTRAN) by using the following conceptual frameworks (also called world views, simulation strategies, and formalisms): (1) event scheduling, (2) activity scanning, (3) three-phase approach, and (4) process interaction. Implementation details under each conceptual framework are covered in a high level without being concerned about execution efficiency. The purpose is to reveal the characteristics of the four conceptual frameworks so that the programmer can select and implement one to achieve certain model quality characteristics such as maintainability, reusability, and execution efficiency. A problem is defined for use as an example for illustrating the concepts throughout the paper.

CR Categories and Subject Descriptors: I.6.0 [**Simulation and Modeling**]: General; I.6.1 [**Simulation and Modeling**]: Simulation Theory—*model classification, types of simulation*

Additional Key Words and Phrases: Formalisms, simulation strategies, world views

1. INTRODUCTION

A simulation model is usually implemented by using a simulation programming language such as GPSS, SIMAN, SIMSCRIPT, SIMULA, and SLAM. In some cases, however, the simulationist must use a high-level programming language (HLPL) such as C, Pascal, and FORTRAN for the implementation (coding, programming) of the simulation model due to the: (1) unavailability of a simulation programming language, (2) inapplicability of a simulation programming language for the problem domain, (3) lack of knowledge of the appropriate simulation programming languages, and (4) need for integrating the simulation model with another software system.

Programming a simulation model in a HLPL is a difficult task. The logic being represented in a simulation program is concurrent in nature. For example, in simulating a traffic intersection we must represent many simultaneous activities such as arrivals of vehicles from several lanes, pedestrian movements from several directions, light changes, and movements of the vehicles through the intersection. On the other hand, computer is a sequential device (unless it has parallel processing capability) and it executes its instructions sequentially. Hence, processing the logic in a sequential manner but yet preserving the concurrency of the activities in the system being simulated is what makes simulation programming in a HLPL difficult.

A *Conceptual Framework* (CF), also called *world view*, *simulation strategy*, or *formalism*, is a structure of concepts and views under which the simulationist is guided for the development of a simulation model. It is vitally important that a programmer follows a CF in implementing a simulation model in a HLPL. The use of a CF facilitates the implementation and significantly reduces its complexity.

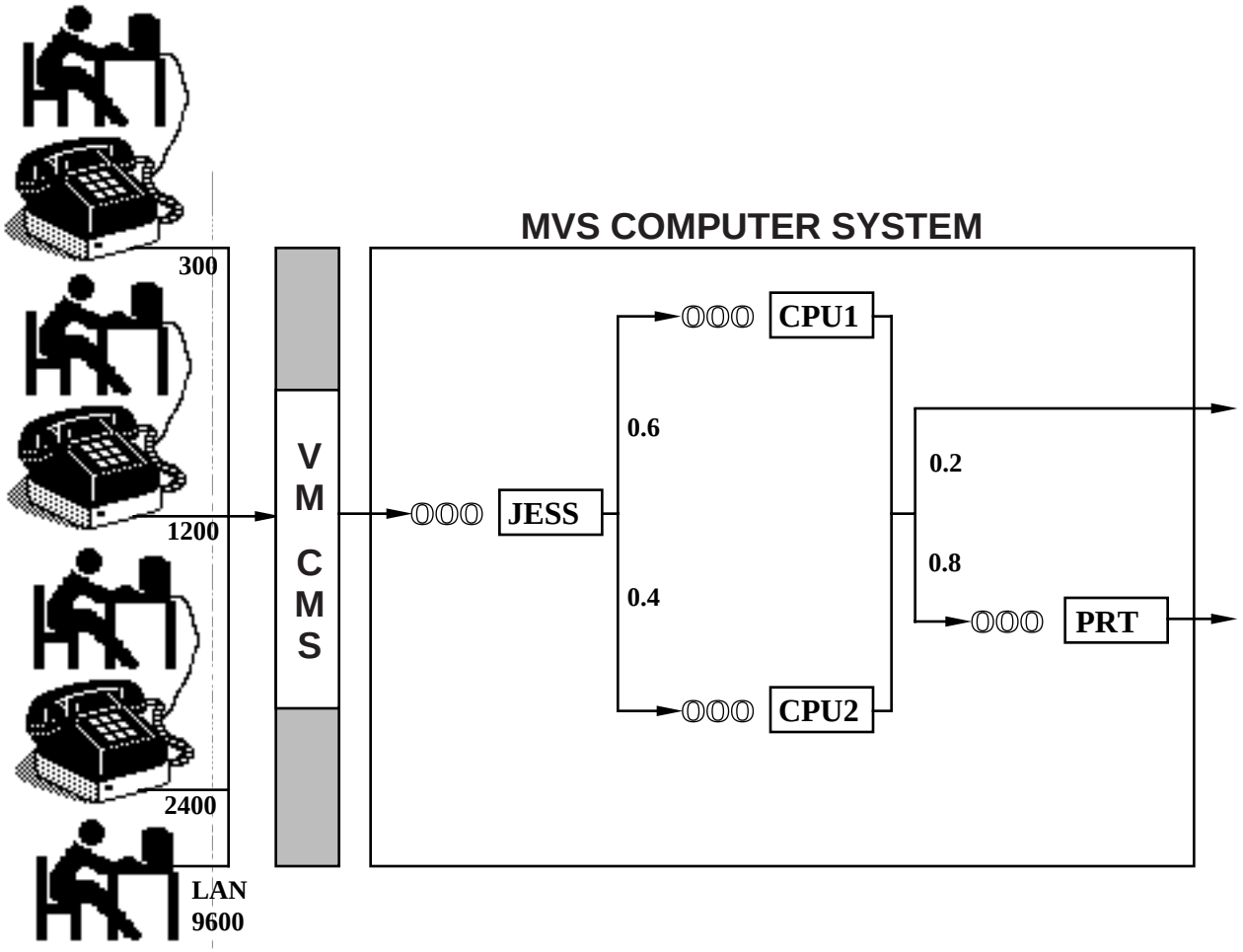
The objective of this tutorial paper is to show how to implement a simulation model in a HLPL by using the following CFs: (1) event scheduling, (2) activity scanning, (3) three-phase approach, and (4) process interaction. Our intent is to cover the implementation details under each CF in a high level without being concerned about execution efficiency. The purpose is to reveal the characteristics of the four CFs so that the programmer can select and implement one to achieve certain model quality characteristics such as maintainability, reusability, and execution efficiency. After providing some background information, a section is devoted for each CF before concluding remarks are given.

2. BACKGROUND

In Section 2.1, we define a problem for use as an example for illustrating the concepts throughout the paper. Section 2.2 provides the definitions of important terms. Section 2.3 introduces the time flow mechanisms.

2.1 Definition of an Example Problem

A Multiple Virtual Storage (MVS) batch computer system operates with two Central Processing Units (CPUs). Users submit their batch programs to the MVS by using the *submit* command on an interactive Virtual Memory (VM) computer system running under the CMS operating system. As shown in Figure 1, the users of MVS via VM/CMS are classified into four categories: (1) users dialed in by using a modem with 300 baud rate, (2) users dialed in by using a modem with 1200 baud rate, (3) users dialed in by using a modem with 2400 baud rate, and (4) users connected to the Local Area Network (LAN)



with 9600 baud rate. Each user develops the batch program on the VM/CMS computer system and submits it to the MVS for processing. Based on the collected data, assume that the interarrival times of batch programs to the MVS with respect to each user type are determined to have an exponential probability distribution with corresponding means as shown in Table 1.

Table 1

Type of User	*Interarrival Times*	*Mean*
Modem 300 User	Exponential	3200 Seconds
Modem 1200 User	Exponential	640 Seconds
Modem 2400 User	Exponential	1600 Seconds
LAN 9600 User	Exponential	266.67 Seconds

A batch program submitted first goes to the Job Entry Subsystem (JES) of MVS. The JES Scheduler (JESS) assigns the program to processor 1 (CPU1) with a probability of 0.6 or to processor 2 (CPU2) with a probability of 0.4. At the completion of program execution on a CPU, the program's output is sent to the user's virtual reader on the VM/CMS with a probability of 0.2 or to the printer (PRT) with a probability of 0.8. Assume that all queues in the MVS computer system are handled by the first-come-first-served discipline and each facility (i.e., JESS, CPU1, CPU2, or PRT) processes programs one at a time. The probability distribution of the processing times of each program is given in Table 2 for each facility.

Table 2

Facility	*Processing Times*	*Mean*
JES Scheduler	Exponential	112 Seconds
Processor 1 (CPU1)	Exponential	226.67 Seconds
Processor 2 (CPU2)	Exponential	300 Seconds
Printer (PRT)	Exponential	160 Seconds

Using a high-level programming language of your choice, write a computer program to simulate the behavior of the MVS computer system. Assuming that the simulation model reaches the steady-state conditions after 3,000 programs, simulate the system for 15,000 programs in steady state and construct confidence intervals for the following performance measures of interest:

[1] Utilization of the JESS (ρ_{JESS}).

[2] Utilization of CPU 1 (ρ_{CPU1}).

[3] Utilization of CPU 2 (ρ_{CPU2}).

[4] Utilization of the Printer (ρ_{PRT}).

[5] Average time spent by a batch program in the MVS computer system (W).

[6] Average number of batch programs in the MVS computer system (L).

2.2 Definitions of Terms

A system can be described in terms of objects (entities), attributes, events, activities, and processes. An *object* (or *entity*) denotes an element of interest in the system. A batch program, the JES scheduler, CPUs, and the printer are the objects in the example problem in Section 2.1. An *attribute* denotes a property of an object or the system, or conveys information about an aspect of an object or the system. In the example problem, system entry time is an attribute of the batch program object; status (showing idle or busy) is an attribute of the JESS, CPU1, CPU2, and PRT objects; and the number of batch programs in the system and the system time are attributes of the system.

An *event* is anything that causes a change in the state of an object and/or in the state of the system. The *state of an object* is defined by the values of all attributes of that object at a particular instant of time. The *state of the system* is defined by the values of all attributes of the system and the values of all attributes of all objects of the system at a particular instant of time. The events of interest of the example problem are shown in Figure 2. Table 3 describes each event and specifies the name of an attribute the value of which is changed by the occurrence of that event.

An *activity* is what transforms the state of an object over a period of time. An activity is initiated by the occurrence of an event and is ended by the occurrence of another event. The activities of interest of the example problem are depicted in Figure 2 and described in Table 4.

A *process* is a sequence of activities or events ordered on time. The processes of interest of the example problem, denoted by P_1, P_2, P_3, and P_4 in Figure 2, are the services provided at the job entry, CPU1, CPU2, and PRT subsystems, respectively.

2.3 Time Flow Mechanisms

Time is a crucial attribute of any system which contains time-varying relationships. The system time is called the *real time*. In a simulation, it is called the *simulated time* and is represented by the simulation clock. The simulated time must not be confused with the real time. The simulated time: (1) is

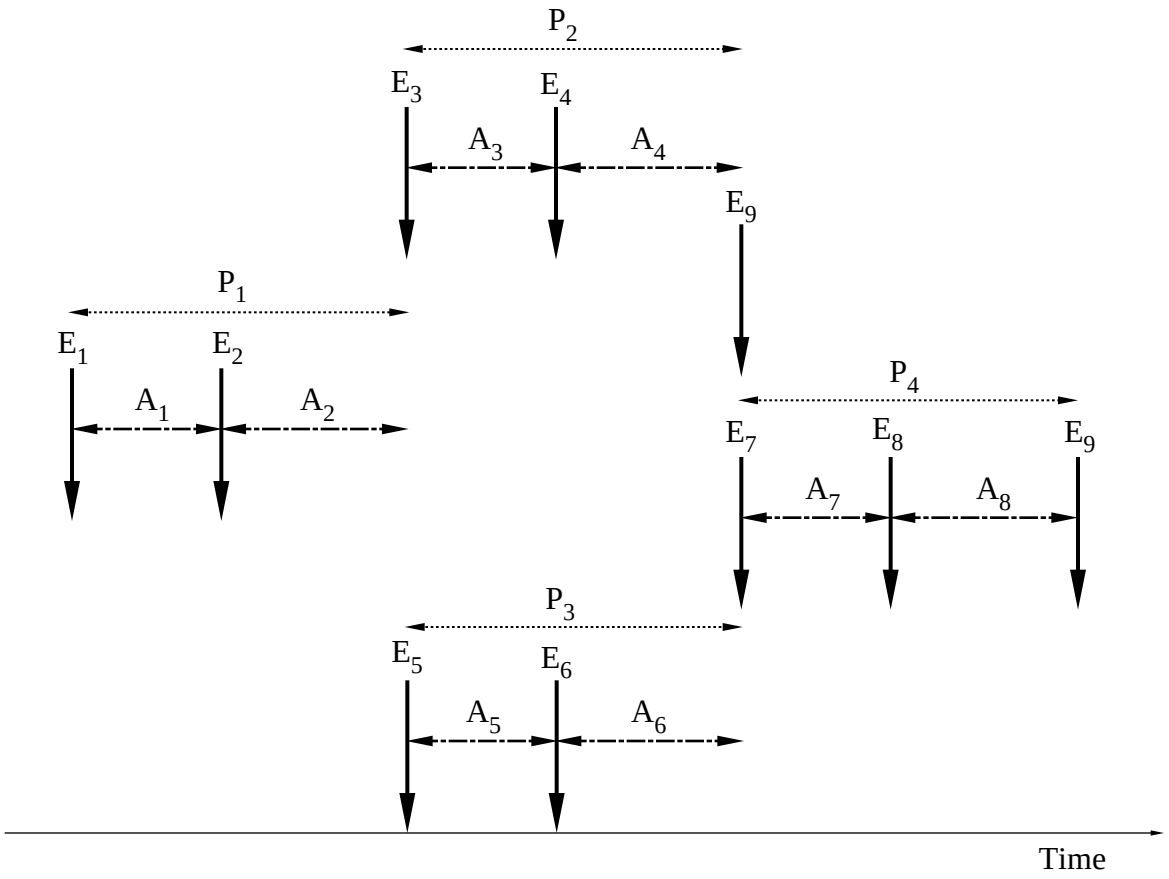


Table 3		
Event	***Description***	***Attribute Changed***
E_1	Arrival to the Job Entry Subsystem	Number of batch programs in the system
E_2	Start of scheduling	Status of JESS
E_3	Arrival to CPU1 subsystem	Status of JESS
E_4	Start of execution on CPU1	Status of CPU1
E_5	Arrival to CPU2 subsystem	Status of JESS
E_6	Start of execution on CPU2	Status of CPU2
E_7	Arrival to printer subsystem	Status of CPU1 or Status of CPU2
E_8	Start of printing	Status of the printer
E_9	Departure from the system	Number of batch programs in the system

Table 4			
Activity	***Description***	***Starting Event***	***Ending Event***
A_1	Waiting for JESS	E_1	E_2
A_2	Scheduling	E_2	E_3 or E_5
A_3	Waiting for CPU1	E_3	E_4
A_4	Execution on CPU1	E_4	E_7 or E_9
A_5	Waiting for CPU2	E_5	E_6
A_6	Execution on CPU2	E_6	E_7 or E_9
A_7	Waiting for the printer	E_7	E_8
A_8	Printing	E_8	E_9

mostly initialized with a value of zero corresponding to any real time value, (2) *always* advances and is never decremented similar to the real time, and (3) can be stopped unlike the real time.

A Time Flow Mechanism (TFM) is a strategy by which the simulated time (simulation clock) is incremented from one value to another. There are two basic strategies for time advancement in a simulation: fixed-time increment and variable-time increment.

In *fixed-time increment TFM*, the time is advanced by a fixed length of t. The simulation clock is changed from t to (t+ t). All state changes occurring during the time interval of t to (t+ t) are processed. The selection of t is vitally important for the accuracy and execution speed of the simulation. A too large t value will invalidate the simulation and a too small t value will slow down its execution. Therefore, the value of t must be carefully chosen depending upon the problem domain. In simulating the U.S. economy, a day may be an appropriate value for t. On the other hand, in the simulation of a computer system a t value of a millisecond may be suitable.

In *variable-time increment TFM* (also called *next-event TFM*), the time is advanced from one state change (event) to another (event). At a particular time t, all state changes (events) are processed. Since the unconditional state changes (e.g., arrival of an object, service completion, departure of an object) can be scheduled in advance, the time of the most imminent state change after time t can be determined and assigned as the next value of the simulation clock.

In variable-time increment TFM, no execution time is wasted for time advancement and for searching state changes to process during those time intervals of no state change unlike the fixed-time increment TFM. On the other hand, sequencing of unconditional state changes is required for the variable-time increment TFM but not for the other. The decision of which one to select must be made in view of the problem domain. If, in a system, a state change occurs every t time units, the fixed-time increment TFM might produce faster execution. For example, in the simulation of a computer operating system, if

a network process is executed every millisecond to find out if a mail message arrived and the simulation time unit is chosen to be in milliseconds, then the fixed-time increment TFM should be selected. For most systems, however, a state change occurs at random times and the variable-time increment TFM proves to be more execution efficient. On the other hand, execution efficiency may not be the number one goal of a simulation study and the fixed-time increment TFM may still be chosen just for the sake of easy implementation of the simulation model.

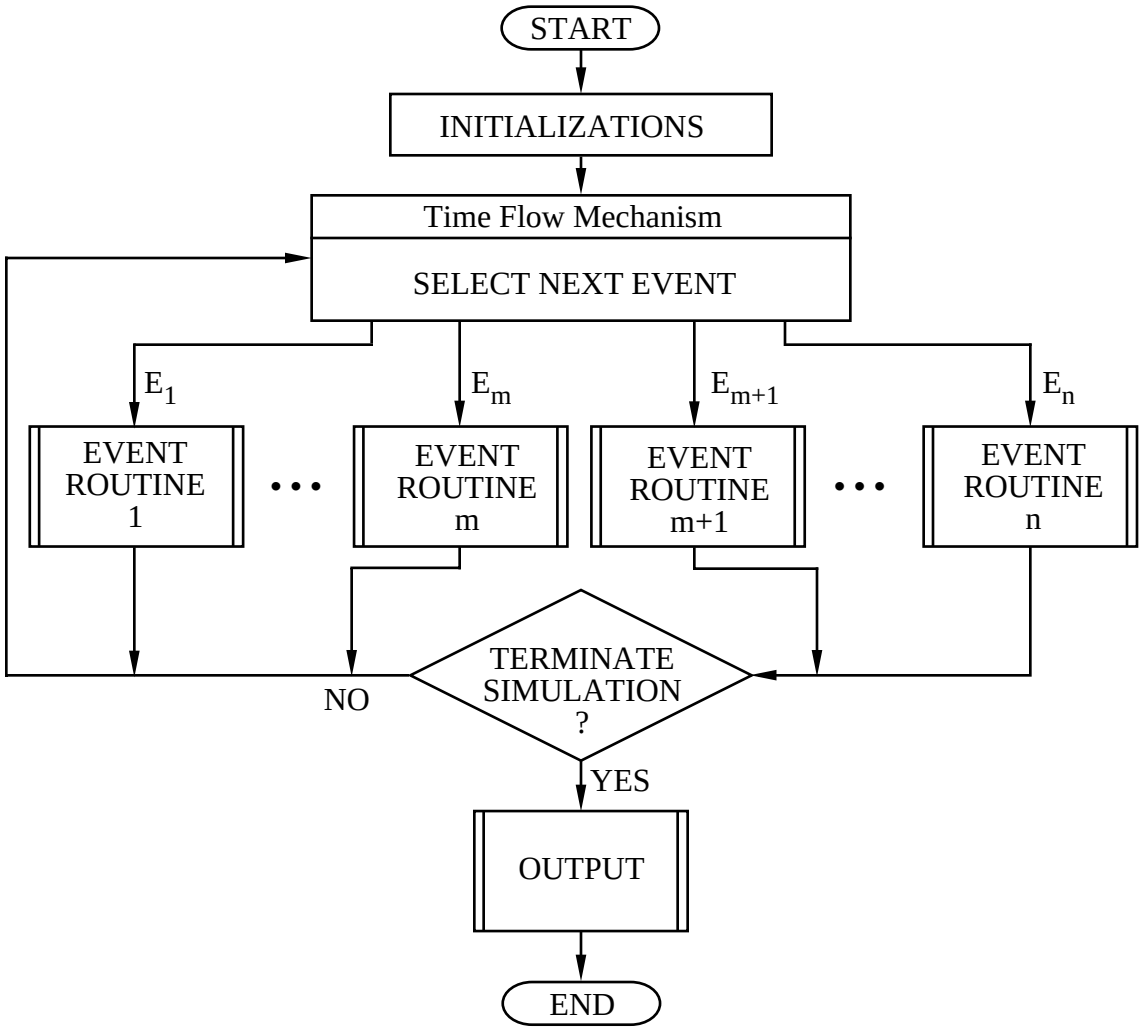
3. EVENT SCHEDULING

Under this CF, an *event* is the major focus of modeling a system. The modeler follows the steps described below:

(1) Identify the objects and their attributes.
(2) Identify the attributes of the system.
(3) Define what causes a change in the value of an attribute as an event.
(4) Write a subroutine to execute each event.
(5) Follow the logic of the flow chart in Figure 3 and use the event list as structured in Figure 4 to develop the simulation program.

The objects, attributes, and events are identified based on the purpose of the study. In the example problem, many objects of the MVS computer system (e.g., disks, I/O channels, tape drives) are excluded from the study for the purpose of further abstracting the reality and building a more abstract and simpler model. If, however, our objective were to build a more detailed model, then we would have included many other objects in our model. A model is built for a specific purpose and its representa-tiveness is judged with respect to that purpose.

The initializations in Figure 3 include the assignment of initial values to all the attributes and the initialization of the event list at (simulated) time zero. Assuming that the simulation starts representing an empty system, the event list is initialized by the first possible events. In the example problem, the first batch program submission from each type of user is scheduled as an E_1 event and merged into the event list with respect to its occurrence time. The occurrence (arrival) time is the interarrival time sampled from the probability distribution since the simulation clock is zero. Four records are created, one for each user type, as a result of the initialization of the event list. Two more attributes need to be recorded for E_1: user type and arrival time. User type is needed to identify which probability distribution to sample from to determine the arrival time of the next batch program to be submitted from that user. Arrival time is needed to calculate the waiting time of the batch program in the system when event E_9 occurs.



	Attribute 1 **Event Occurrence Time**	*Attribute 2* **Event Identification**	*Attribute 3*	• • •	*Attribute k*
Record 1				• • •	
Record 2				• • •	
Record 3				• • •	
	⋮	⋮	⋮	⋮	⋮
Record r				• • •	

Using either the variable-time or fixed-time increment TFM, the next event to be executed is determined and is executed. After the execution of some events simulation termination condition must be tested. In the example problem, the termination condition is defined in terms of the number of departures from the system. Therefore, E_9 is the only event after the execution of which the simulation termination condition must be tested. If the condition is satisfied the simulation output is produced and the simulation ends.

To facilitate the selection and processing of events, the event list in Figure 4 can be used. Each event is described by a record containing the event's attributes. Occurrence time and identification are the two required attributes. The records (events) are sorted with respect to the occurrence time in ascending order. Therefore, in using the variable-time increment TFM, the event on the top of the list is always the next event to execute. The simulation clock is updated to the occurrence time of the next event. After the event execution, record 1 is deleted and all the other records are moved up by one. Execution of an event may produce another one which is merged into the event list with respect to its occurrence time. The new event on the top of the list becomes the next event to execute. This cycle continues until the termination condition holds true.

4. ACTIVITY SCANNING

Activity scanning (also known as the two-phase approach) was first used in the programming language CSL [Buxton and Laski 1962]. It is a state-based approach to simulation modeling. *Activity* is the basic building block of this CF. Activity scanning is similar to the rule-based programming, commonly used in Artificial Intelligence, in which a rule is specified upon the satisfaction of which a predetermined set of operations are performed. Under the activity scanning CF, the modeler describes an activity in two parts:

Condition: A condition or a compound condition (formed by AND, OR, and NOT logical operators) which must be satisfied in order for the activity to take place.

Actions: The operations of the activity performed upon the satisfaction of the activity's condition.

In the example problem, the condition of activity A_1 is specified as: "Is time equal to AT?" where AT is the arrival time of a batch program. The actions of A_1 would be to: (1) add the batch program to the JESS queue, (2) increment the number of batch programs in the system by one, (3) generate the arrival time of the next batch program, (4) create another A_1 activity with the new arrival time. The condition of activity A_2 is specified as: "Is the JESS idle AND does the JESS queue exists?" The actions of A_2 would be to: (1) change JESS status to "busy", (2) remove the batch program from the queue, (3) sample

a processing time (PT) from the probability distribution, (4) add PT to current time to determine end of processing time (EOPT), (5) select CPU 1 or 2 probabilistically, and (6) create activity A_3 or A_5 depending upon the CPU selected. The condition of A_3 is specified as: "Is time equal to EOPT?" The actions of A_3 would be to: (1) change JESS status to "idle" and (2) add the batch program to the CPU1 queue.

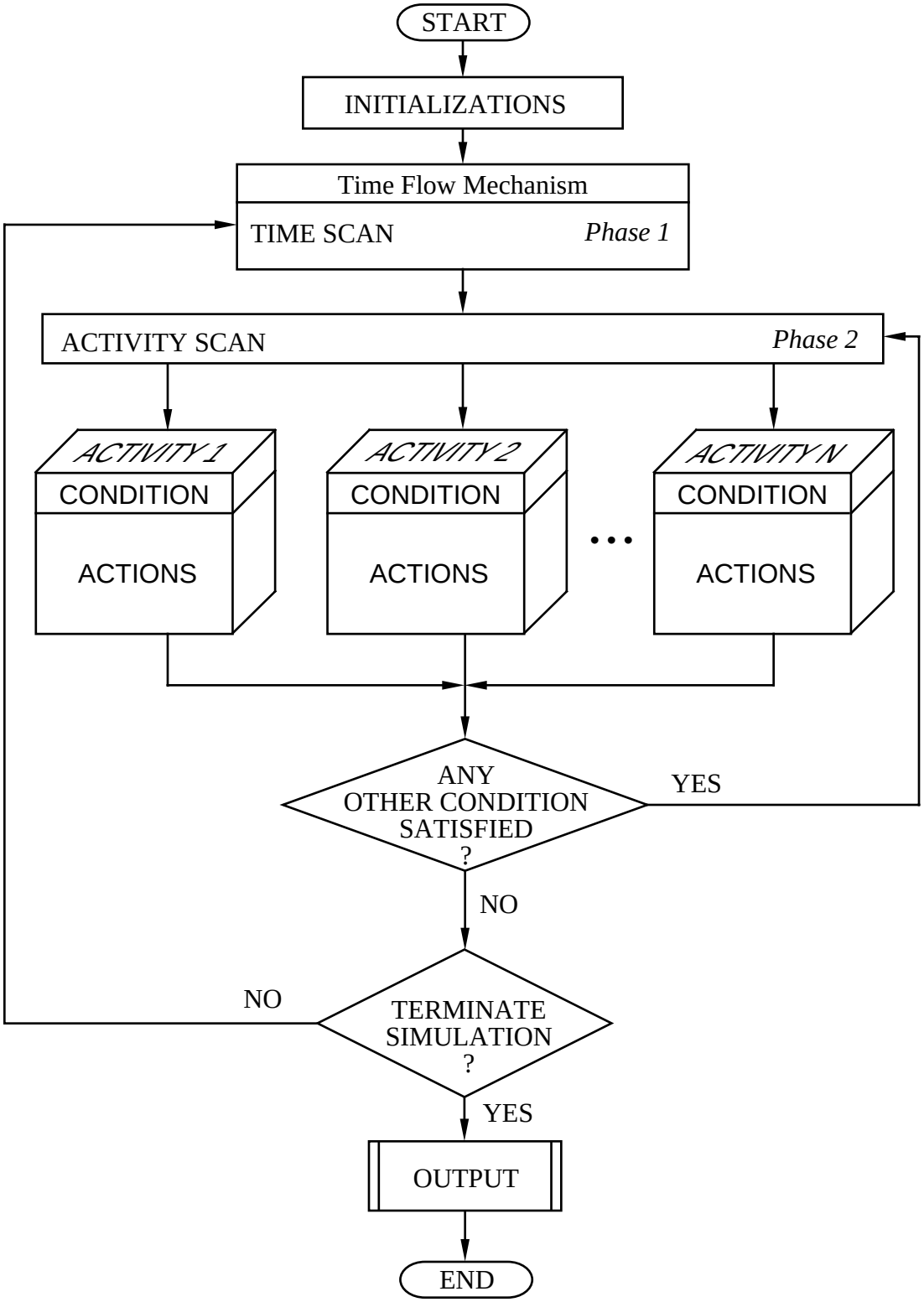
Figure 5 shows the overall logic of activity scanning. The initializations include the assignment of initial values to all the attributes and the creation of four A_1 activities corresponding to the arrivals of first programs submitted from the four types of users. The fixed-time increment TFM is the one used in the purest implementation of activity scanning. In phase 1, the time is advanced by the fixed t amount from t to (t+ t) and phase 2 is conducted with a simulation clock time of t = (t+ t).

In phase 2, the conditions of activities are tested in the order of activity priorities (not in the order of activity numbers). If an activity's condition is satisfied, the actions of that activity are performed. Testing all the conditions and performing the actions corresponding to the satisfied conditions constitute a single scan. A single scan may not be sufficient. Some actions performed may cause the satisfaction of earlier unsatisfied conditions. This requires the restart of the scan. All conditions must be repeatedly tested until no condition is satisfied at the current simulation clock time.

Activities should be prioritized for the order of condition testing by the modeler depending upon the problem domain. For instance, in the example problem, assume that the JESS is idle and its queue is empty at time t. Suppose that there is an arrival to the JES also at time t. Now, A_2's condition will be unsatisfied if it is tested before A_1's and will be satisfied if it is tested after A_1's. Thus a rescan is caused by the first case but not by the second. Hence, the order of testing the conditions is important.

Notice that Figure 5 shows only the N different activities (N=8 in the example problem) for the sake of simplicity excluding the occurrences of the same activity under different conditions. For example, there are four activities of type A_1 with four different conditions: "Is time equal to AT_j ?", where AT_j represents the arrival time of a batch program submitted by the user of type j.

Activity scanning CF produces a simulation program composed of independent modules waiting to be executed. Due to the need to scan the conditions repeatedly and in most cases because of the fixed-time increment TFM, the simulation runs slowly. However, the simulation program is modular, maintainable, easy to modify, easy to implement, and easy to understand. Considering the fact that about 70% of software development cost is attributed to maintenance, achievement of maintainability should not be undervalued. In those simulation studies where the maintainability and ease of implementation are much more important than the execution efficiency, activity scanning CF would be a very good choice.


START
INITIALIZATIONS
Time Flow Mechanism
TIME SCAN
Phase 1
ACTIVITY SCAN
Phase 2
ACTIVITY 1
CONDITION
ACTIONS
ACTIVITY 2
CONDITION
ACTIONS
ACTIVITY N
CONDITION
ACTIONS
ANY OTHER CONDITION SATISFIED ?
YES
NO
TERMINATE SIMULATION ?
NO
YES
OUTPUT
END

5. THREE-PHASE APPROACH

In order to try to remedy the execution inefficiency of activity scanning, Tocher [1963] suggested the three-phase approach a year after activity scanning was first used in the Control and Simulation Language [Buxton and Laski 1962]. During those years, computer time was very expensive and the execution efficiency was extremely important.

The three-phase approach combines activity scanning and event scheduling CFs. It provides a mixture of the state-based and time-based approaches to simulation modeling. Events and activities are the two basic building blocks of modeling; however, events are labeled as activities of duration zero.

Activities are classified into two categories [Pidd 1984]:

B Activities: are the bound-to-occur or book-keeping activities that represent the unconditional state changes (unconditional events) which can be scheduled in advance (e.g., E_1, E_3, E_5, E_7, and E_9 in the example problem).

C Activities: are the conditional or co-operative activities that represent the state changes which are conditional upon the co-operation of different objects or the satisfaction of specific (compound) conditions (e.g., A_2, A_4, A_6, and A_8 in the example problem).

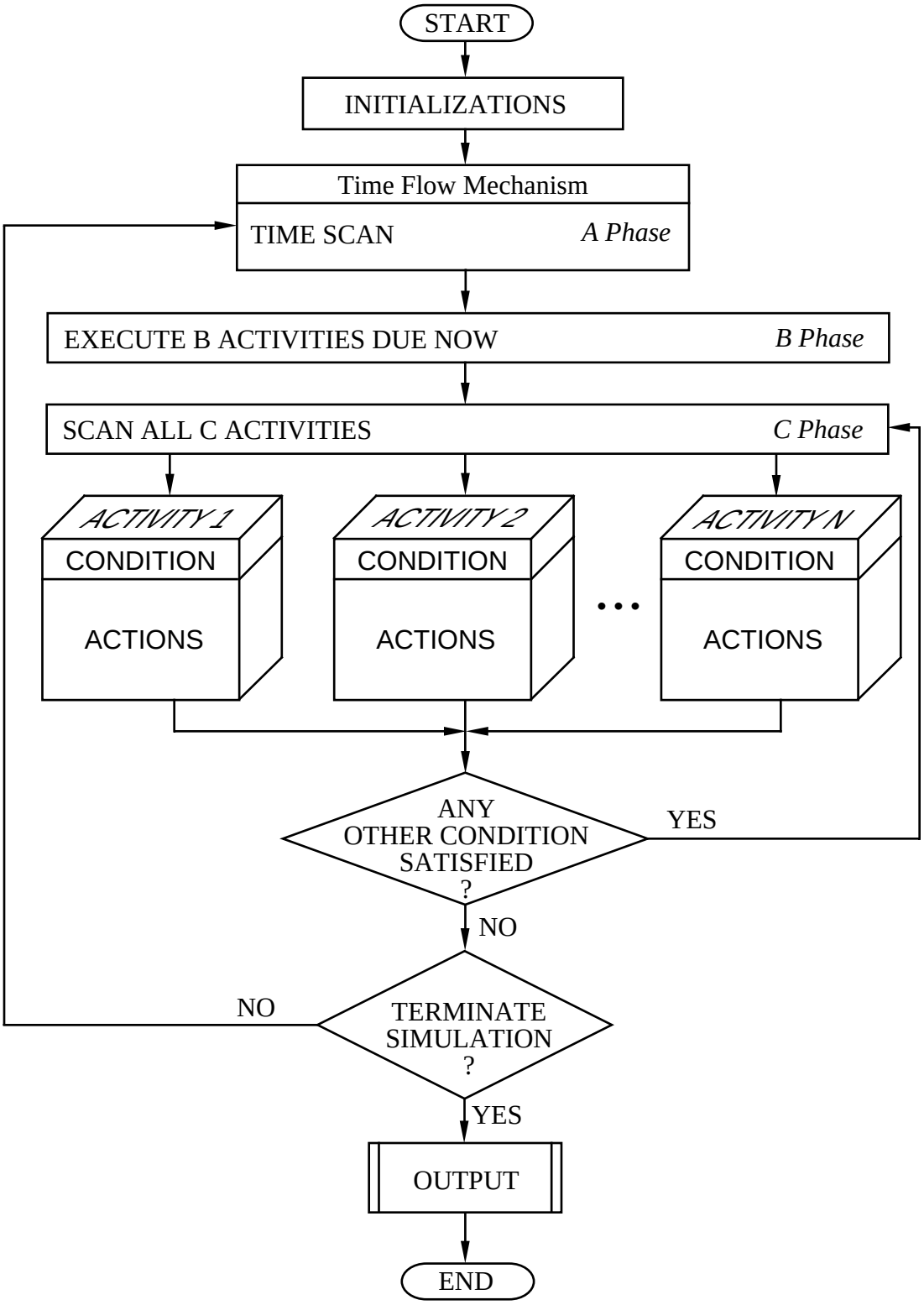
There are two major differences between activity scanning (Figure 5) and the three-phase approach (Figure 6):

(1) The TFM can implement the variable-time increment strategy. Knowing the occurrence times of B activities (unconditional events) in advance, the most imminent event can be determined and its occurrence time can be assigned as the new value of the simulation clock in Phase A.

(2) All B activities (unconditional events) with occurrence times equal to the simulation clock value are executed in Phase B.

The third phase, Phase C, of the three-phase approach works in the same way as in the activity scanning except that the activities being scanned are C activities only.

Both of the major changes improve the execution efficiency; however, this improvement is achieved at the expense of some conceptualization difficulties. Some complexity of event scheduling is inherited. The complexity of conceptualization is increased by mixing two radically different approaches, namely, stated-based and time-based approaches.


START
INITIALIZATIONS
Time Flow Mechanism
TIME SCAN
A Phase
EXECUTE B ACTIVITIES DUE NOW
B Phase
SCAN ALL C ACTIVITIES
C Phase
ACTIVITY 1
CONDITION
ACTIONS
ACTIVITY 2
CONDITION
ACTIONS
ACTIVITY N
CONDITION
ACTIONS
ANY OTHER CONDITION SATISFIED ?
YES
NO
TERMINATE SIMULATION ?
NO
YES
OUTPUT
END

6. PROCESS INTERACTION

Under this CF, a modeler describes the life cycle of an object which moves through and interacts with the processes of the system under study. Object and process descriptions constitute the underpinnings of this approach.

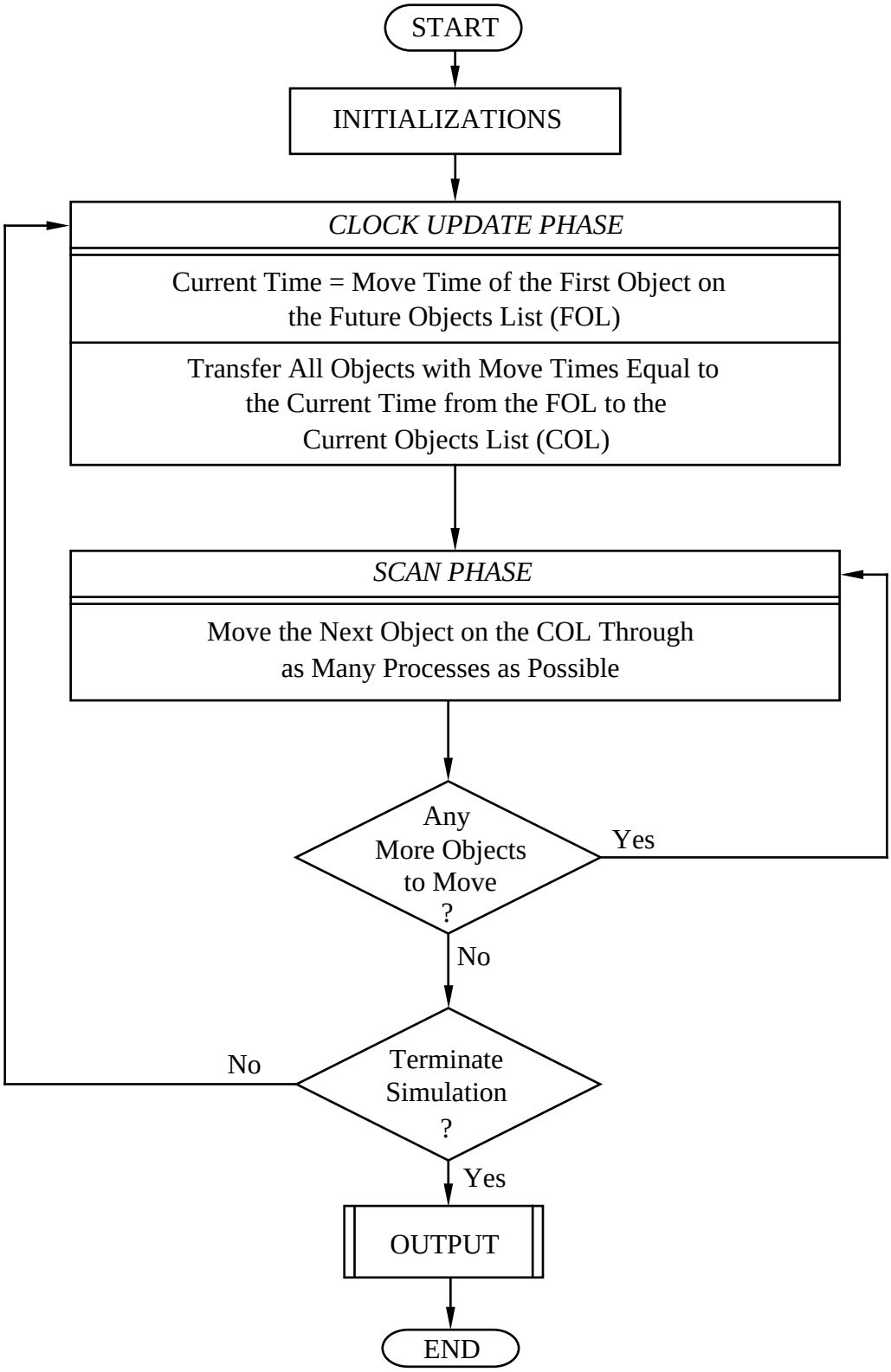
An object is classified into two: dynamic and static. A *dynamic object* is the one which comes into the model, logically moves through some processes, and leaves the model (e.g., the batch program in the example problem). A *static object* is the one which does not logically move (e.g., JESS, CPUs, and PRT in the example problem). Objects are described in terms of attributes. Identification, move time, current location, next location to move to, and priority level are the five attributes commonly recorded for each dynamic object.

Figure 7 shows the implementation logic in a similar manner used in GPSS. There are four phases of the logic: initializations, clock update, scan, and output. After initializations, the simulation is conducted by going back and forth between the clock update and scan phases. Output phase is executed at the termination of the simulation.

In the initializations, all attributes are initialized and the first dynamic objects are created and placed on the Future Objects List (FOL) in ascending order of their move times. In the example problem, four dynamic objects are created to represent the first batch programs submitted by the four types of users. The user type also needs to be recorded as an attribute for each dynamic object.

In the clock update phase, the time is advanced to the move-time of the front-end object of the FOL. Note that this move-time is the smallest since the dynamic objects are placed on the FOL always in ascending order of their move times. All dynamic objects with move-times equal to the current simulation clock value are transferred from the FOL to the Current Objects List (COL). The objects are placed on the COL always in the order of their priority levels and their move-times are changed to As Soon As Possible (ASAP). First-come first-served within a priority level is a commonly used queue discipline. Thus, an incoming object is placed on the COL as the last member of the queue within its priority level.

In the scan phase, the objects on the COL are moved, one by one, through as many processes as possible in the order of their placement (priority levels). When the dynamic object is set into motion, one of the following can happen to stop its movement: (1) the object faces an unsatisfied condition, (2) the object is deliberately delayed for a while (e.g., going into service), (3) the object dies or leaves the model, and (4) the object is deliberately stopped for some reason. The movement of an object can result in state changes and can make it possible to move an earlier object which couldn't be moved. For example, suppose that there is a batch program (dynamic object 1) in the queue of CPU1 which is finishing the execution of another batch program (dynamic object 2) at time t. Assume that object 1 is positioned


START
INITIALIZATIONS
CLOCK UPDATE PHASE
Current Time = Move Time of the First Object on the Future Objects List (FOL)
Transfer All Objects with Move Times Equal to the Current Time from the FOL to the Current Objects List (COL)
SCAN PHASE
Move the Next Object on the COL Through as Many Processes as Possible
Any More Objects to Move ?
Yes
No
Terminate Simulation ?
No
Yes
OUTPUT
END

on the COL before object 2. The attempt to move object 1 into CPU1 will fail since the CPU1 is busy. Object 2 can be moved making CPU1 idle. Situations similar to this one will require restarting the scan of objects. Therefore, the COL must be repeatedly scanned until no more objects can be moved.

Note that whenever a dynamic object enters into the model, its movement is temporarily stopped to schedule the arrival of the next dynamic object from the corresponding source. In the example problem, whenever a batch program arrives at the job entry subsystem, a new object is immediately created with an arrival (move) time equal to the current simulation time plus the interarrival time randomly sampled from the probability distribution corresponding to the value of the user type attribute of the current object. The new object is then placed on the FOL and the movement of the entering object is resumed.

7. CONCLUDING REMARKS

Development of a simulation program in a HLPL is certainly a difficult task especially for large scale and complex systems. One of the four CFs described in this paper can be employed to reduce the complexity of the implementation. A CF should be chosen by considering the problem domain and the selected model quality characteristics (e.g., maintainability, execution efficiency, modifiability, reusability, ease of development).

Unfortunately, it is not possible to achieve some model quality characteristics together since they conflict with each other. For example, a model which is maintainable runs slower because of the overhead caused by the maintainability characteristic. On the other hand, a model can be developed as tightly integrated running very fast but being very difficult to maintain. Hence, a trade-off need to be made among conflicting quality characteristics in view of the study objectives.

Execution efficiency had generally been the number one goal in simulation model development during 1960s and 1970s. Today, human time is usually much more expensive than computer time and in some simulation studies, other model quality characteristics such as maintainability, modifiability, and reusability are given higher priority over the execution efficiency.

The reader is recommended to develop four simulation models of the example problem in Section 2.1 in a HLPL by using the four CFs described herein. To aid in the validation of the simulation models, the true values of the six performance measures are given below:

[1] Utilization of the JESS (ρ_{JESS}) = 0.70

[2] Utilization of CPU 1 (ρ_{CPU1}) = 0.85

[3] Utilization of CPU 2 (ρ_{CPU2}) = 0.75

[4] Utilization of the Printer (ρ_{PRT}) = 0.80

[5] Average time spent by a batch program in the MVS computer system (W) = 2400 seconds

[6] Average number of batch programs in the MVS computer system (L) = 15

ACKNOWLEDGMENTS

This research was sponsored in part by the U.S. Navy under contract N60921-83-G-A165-B03 through the Systems Research Center at VPI&SU. Stimulating discussions with David W. Balmer, E. Joseph Derrick, Richard E. Nance, and Robert M. O'Keefe are gratefully acknowledged.

BIBLIOGRAPHY

Buxton, J.N. and J.G. Laski (1962), "Control and Simulation Language," *The Computer Journal 5*, 194-199.

Clementson, A.T. (1966), "Extended Control and Simulation Language," *The Computer Journal 9*, 3 (Nov.), 215-220.

Crookes, J.G., D.W. Balmer, S.T. Chew, and R.J. Paul (1986), "A Three-Phase Simulation System Written in Pascal," *Journal of Operational Research Society 37*, 6 (June), 603-618.

Davies, R. and R.M. O'Keefe (1987), *Simulation Modelling with Pascal*, Prentice-Hall, U.K.

Fishman, G.S. (1973), *Concepts and Methods in Discrete Event Digital Simulation*, John Wiley & Sons, New York.

Franta, W.R. (1977), *The Process View of Simulation*, North Holland, Amsterdam, Holland.

Hooper, J.W. (1986), "Strategy-Related Characteristics of Discrete-Event Languages and Models," *Simulation 46*, 4 (Apr.), 153-159.

Nance, R.E. (1971), "On Time Flow Mechanisms for Discrete System Simulation," *Management Science 18*, 1 (Sept.), 59-73.

Nance, R.E. (1981), "The Time and State Relationships in Simulation Modeling," *Communications of the ACM 24*, 4 (Apr.), 173-179.

Pidd, M. (1984), *Computer Simulation in Management Science*, John Wiley & Sons, Chichester.

Tocher, K.D. (1963), *The Art of Simulation*, English Universities Press, London.