# A LANGUAGE ORIENTATION FOR THE DISCRETE EVENT MODELING OF APPLICATION PROGRAMS AND OPERATING SYSTEMS

Leo J. Cohen

Performance Development Corporation

## FORWORD

Discrete event simulation of computer programs, particularly operating system programs, has until recently been an elusive objective of computer performance simulation. However, out of my own experience and my observations of the efforts of others, I have come to believe that the most direct approach to achieving such a simulation capability is essentially linquistic.

By itself, a lanquage description for program modeling makes little sense if it is presented outside of the context of suitable techniques for creating the simulation environment. That is, an intimate relationship must exist between the simulation language necessary for the precision representation of application and operating system programs, and certain fundamental properties that simulators of computer systems must have. For this reason my discussion begins with a distinctive characterization of jobs and programs, showing that these are not only separate entities in the real computer but must be represented as such in the simulated system as well. Furthermore, the lanquage requirements can only be appreciated in the context of simulator operations, and therefore these general techniques, under the headings "Executions", "Interrupts", and "Creating Simulated Time", are presented.

I consider these subjects to be basic to any appreciation of the necessary aspects of a linquistic capability, and with these concepts in hand, I then turn to discussion of an explicit language orientation. This is broken down into two major categories, the first for representing the logic of a program and the second for representing the loads that this logic imposes on the system.

The detailed simulation of application and operating system programs can be made precise in both the logic and the load dimensions if capability for manipulation of actual data by the simulation models is provided. This requires memory storage for actual data on the part of the model and is a major subject addressed. Then, as a natural consequence of both the linguistic orientation and memory capability, I am able to extend the basic concepts to allow for the precise representation of both reentrant and recursive functions.

## BASIC CONCEPTS

### Jobs

In the descrete event environment we must identify activities which, when actuated or operated by a transaction create an explicit event in time. In simulating the performance of computing systems, and particularly the execution of programs, one such activity is the execution of instructions. The event created is, of course, the load produced on the CPU mechanism of the computer. In the real machine jobs produce such loads. We therefore will define the "job transaction", saying that a simulated event is created when a job transaction passes through a simulated instruction activity.

As a simple example of a job/event combination consider the following simulated instruction activity:

COMPUTE 1000.

When operated by a job transaction this instruction produces a time delay in the progress of the simulated job that is equal to the execution time required for 1000 instructions in the simulated computer. This is equivalent to advancing simulated time by this interval as well.

In an actual computing system the job has an identity and represents a collection of data. For example, in an OS/MFT environment a job that has been allocated core memory is identified by its task control block number, and the task control block and all appended data make

up the data entity for the job. A more general treatment and broad discussion of the job as an identifiable data entity within the computing system is contained in (1). For the present discussion we may assume that each job is uniquely identifiable and is associated with a set of fields in which data may be stored and from which such data may be retrieved. In this sense, then, the job transaction under consideration is virtually identical to the GPSS transaction. That is, the job transaction may be thought of as an identification plus a set of fields which together form a record that then moves over the flowchart of our model, where each entry in the flowchart represents an activity. As such a job transaction crosses an activity a corresponding event is created.

## Program Models

A computer program has a flowchart, which we may think of as a progression of detail from the highest level of process description down to a collection of encoded statements. That is, a flowchart may be atomic in varying degrees. Every flowchart, no matter what its detail, represents the collection of activities which may create events in a computer, with the possibility of event creation expressed by the logic of the flowchart. To put this somewhat differently, the flowchart, like the program, is a potential for creating load in a computing system. Furthermore, nothing happens in the real computer until a job is present in the system for the execution of that program.

For simulation purposes we would like to be able to express the flowchart of a real program in a linguistically equivalent way so that by passing a job transaction through that flowchart we might recreate the loads in the sequence in which they were imposed on the real system. This raises the issue, then, of being able to represent the flowchart logic of programs. That is, some instruction activity must be available in the simulation language repertoire that provides a means for logical expression. We shall refer to such instruction activities as the "zero time instructions", as opposed to the "simulated time instructions" which produce time-associated events.

## Executions

The notion of a program model flowchart, each node of which is representative of one or several simulation instruction activities, suggests that the simulation process may be thought of as the movement of the data entity represented by a job transaction from node to node, either in zero time or as delayed by the associated simulated time of certain events. Thus, the zero time instructions will be employed to create the logic of the movement, and the simulated time instructions to express the time reality of the movement. The "execution" of a program in the simulation sense, therefore, is the passing of a job transaction through its model flowchart from an entrance point to an exit point. Functionally speaking, then, we require a simulation instruction repertoire by means of which such model flowcharts may be encoded.

## Application and Operating System Models

Let us suppose for the moment that we have such a simulation instruction repertoire, and that from it we have derived a collection of program models with which to simulate job executions in our real system environment. Let us further suppose that we have a collection of job transactions which are created according to some specified timing rules. The timing rules for creation of a job transaction will be referred to as a "creation set". It is these creation sets that cause jobs to enter our simulation system. Thus we have separated job models from program models and the effective performance of a given job is directly dependent on which of the program models it passes through, and what instruction activities have been represented in those models.

Although we may make a distinction between operating system program models and application program models, aside from the repertoire of instruction activities that are useful to the creation of an operating system model, there should be little difference in conceptual substance between the two. To express this in more concrete terms, let us say that our simulation system, via a creation set, will make available a job transaction designed explicitly for the purpose of executing the operating system program model. This means that our job transactions now divide into two classes, namely application and operating system. When an application job makes reference to the operating system, then in transaction terms its job transaction crosses an "entrance boundary" of the operating system (2), and thus activates the operating system job transaction. Events are then created in the operating system model by this transaction, up to the point where it reaches an "exit boundary" of the operating system. It is from this point that some specified application job transaction (usually

selected by the logic of the operating system model) continues the simulation by moving further through its current program model.

The operating system entrance and exit boundaries can be given an explicit linguistic expression that reflects this change of execution state, or mode, in the real system (3). For simulation, however, this concept has the further advantage of clearly identifying when a simulated execution in the form of a job transaction, has left the confines of an application program model and entered the operating system model. This means that whenever the operating system job transaction is "executing" during the simulation we must be accumulating performance figures that are associated with the simulated operating system and not with any of the simulated application programs.

## Forced Entries

Interrupts in a computing system generally mean that the current execution of a program is intervened so that the CPU can be assigned to execution of a different, higher priority function that is usually associated with the operating system. In the context of the present discussion, however, we may say that the job leaves its present execution and is forced to enter the operating system. We will see that these interrupts fall into the two general categories, synchronous and asynchronous, and that it is useful to refer to them jointly as "forced entries".

The synchronous forced entries are those associated with certain application program instructions, such as READ, and with service request instructions. Each of these instructions forces an entry to an associated point in the operating system, and may be thought of as interrupts that are synchronized with the execution of the application program. As opposed to this, the asynchronous forced entries are derived from I/O termination interrupts, clock interrupts, and general alert interrupts from sources external to the system. When any of these interrupts occur, the particular forced entry action is realized by taking the next instruction for execution from the associated interrupt location within the operating system.

In simulation terms we will say that forced entries move the application job transaction currently associated with the simulated CPU out of its program model and to an operating system model entrance boundary. Here the operating system job transaction takes over. By this means, then, the application job transaction is forced to enter the operating system model.

## Creating Simulated Time

The zero time instructions create no simulated time, and therefore occur instantaneously with respect to the simulated time of the model. When the operating system model has given control of the simulated CPU to a particular job transaction, it is allowed to continue unobstructed in its application model through all zero time instructions until it encounters a simulated time instruction. At this point, computations are made by the simulator system to determine the amount of simulated time involved, and then a "future event" is established for that much time in the simulated future. It is at this time that the associated computation will be complete. Such future events are stored in a "future events chain" in chronological order. Also included in this chain are future events related to I/O terminations, future creation set events bringing new job transactions into the model, and, in fact, events for all of the interrupt conditions currently present in the simulation model.

Suppose, then, that a CPU event has been established for completion at, say, 25 milliseconds in the future. However, due to prior activity in the model, as of this simulated moment a data transfer activity is to terminate within 14 milliseconds. This creates an interrupt event that will be higher in the chain of future events and will therefore occur earlier in future simulated time. The simulator steps, or increments the simulated time clock by 14 milliseconds to this next future event, thus making the present time equal to that of this next future event. The I/O interrupt event is then interpreted, and in general will cause a simulated interrupt of the CPU event which still has 11 milliseconds to go. In an actual computing system the interruption will cause the operating system to take control away from the currently operating job in order to use the CPU for processing the interrupt. In simulation terms this interruption appears as a forced entry of the application job transaction currently simulating 25 milliseconds of CPU activity. Thus, the operating system job transaction is assigned to the CPU and then passes through that portion of the operating system model designed to represent the logical actions to be taken as a consequence of the occurrence of this particular interrupt. The simulator system also observes the amount of the CPU event left to go for the interrupted job and when the operating system model, as a consequence of its logical determinations, later decides to return to the continuation of the interrupted job transaction, it will do so as if a compute instruction requiring 14 milliseconds rather than 25 milliseconds had been executed by the job transaction. That is, a future CPU event representing 11

milliseconds is established for the application job transaction.

A simulation therefore proceeds as follows: Beginning at simulated time zero, an operating system job transaction is introduced to the model. The operating system job transaction then usually goes into an idle state, where it awaits the first creation set introduction of an application job transaction. Simulated time, of course, advances while in the idle state until the first job transaction appears. When it does, there is an interrupt that brings the operation system job transaction out of its idle state and takes it to that portion of the operating system model where acceptance activities for a new job are simulated. In this fashion, job transactions are brought into the simulated system according to the logic of the operating system model, and are ultimately allocated simulated core memory space and set to execution of their respective program models. These job transtransactions execute zero time instructions in their own application models. They also execute instructions that create forced entries and lead to the operating system model and its execution by the associated operating system job transaction. Here, appropriate actions will be taken, leading ultimately to the selection of application job transactions and their continuation in program models. It is in this fashion that the simulation progresses, with the simulated time instructions moving the simulated time forward, thereby creating the loads in the system. In this way, also, simulated time moves from simulated time zero to the final time specified by the user for his simulation run.

## ZERO TIME INSTRUCTIONS

The zero time instructions of a simulation language provide the mechanism for implementing the logic of the program model. When such a capability exists in a simulation system we may think of the programs to be modeled as flowcharts, and then may consider their simulation to be equivalent to the implementation and execution of the flowcharts by the simulator itself. Thus, given a flowchart of some particular programming function, that flowchart has logical structure, and with the zero time instructions we should be able to implement that logic with a suitable degree of approximation. We are then able to simulate the loads imposed by that program via the simulated time instructions, mentioned earlier and discussed below, where these loads are created within the logical framework expressed by the zero time instructions.

It is convenient to break the zero time instructions down into four broad categories. These are, the Application group, the Operating System group, the Interrupt group, and the Common group. Each of these groups are discussed in some detail in what follows. In order to give substance to this discussion, however, it is necessary to refer to a simulator system that implements the orientation suggested in this paper. The Systems Analysis Machine (SAM) is such a system (4). In the instruction repertoire examples that follow, the linguistic forms employed are taken from SAM, and the several figures of SAM instructions that are shown represent a sampling from the complete instruction repertoire. Furthermore, there are other instruction classes and types which represent expanded SAM capabilities that are beyond the general interests of this presentation, and therefore have not been included.

### The Application Group (Figure 1)

The set of zero time instructions associated with the Application group comprise the synchronous forced entry instructions. That is, on their execution in a program model by a job transaction, an entry to the simulated operating system is made. For example, consider the I/O reference instructions in the Application group. In the actual computer the execution of READ in an application program is an indication of a function to be carried out, not the initiation of the function itself. This latter action is usually reserved to some centralized program that is often described as being within the operating system. The simulation function parallels this exactly. When a job transaction passes over an I/O instruction in an application program model, an entry is forced to the simulated operating system where the logic associated with record deblocking, data transfer, queueing, waiting, and so on is simulated. This is accomplished by an operating system job transaction. Then, under circumstances dictatated by the particular operating system model, control is returned at some later time to the application job transaction to continue its simulated execution in the application program model.

The instruction

READ PAYROLL

is a Sam function, which when executed by a job transaction causes the actions described above. The reference is to the simulated file PAYROLL.

The memory reference instructions of

the Application group allow the program model to simulate calls for the expansion or release of working storage. These instructions take a number of forms and may be related to either contiguous or non-contiguous storage that may be associated with either the simulated job space of the job executing the program, or with the program itself. Again, these are forced entry instructions, indicating that the activity involved is to be carried out under the auspicies of the control software of the system.

Application Group (SAM sample)

I/O Reference

CLOSE-IN
CLOSE-OUT
OPEN-IN
OPEN-OUT
READ
WRITE
SEEK

| Program Reference | Memory Reference* |
|---|---|
| CALL-PGM | GET-CJS |
| OPEN-PGM | GET-NCPS |
| CLOSE-PGM | RET-CPS |
| EXIT | RET-NCJS |

*CJS = contiguous job space
CPS = contiguous program space
NCPS = non-contiguous program space
NCJS = non-contiguous job space

FIGURE 1

Similarly, the Program Reference instructions of the application group force entries to the operating system in order to complete sub-program calls, to return from a sub-program to the calling program, to preload and overlay programs, and so on. In the following example,

CALL-PGM XEMPT,

the job transaction enters and passes through the operating system model. In the course of its passage, the logic of the operating system model may determine if the program XEMPT is currently allocated, seized by some other job, etc. That is, the operating system model must execute zero time instructions that represent the actual operations necessary to completing a sub-program call. As a final step, the application job transaction will be passed on to the execution of the program model XEMPT.

## The Operating System Group (Figure 2)

Corresponding to each of the forced entries, both synchronous and asynchronous is a sub-progroup of instructions that is associated with operating system modelling.

The I/O control instructions of the Operating System group are used to test the availability of the various elements in the data transfer chain between core and peripheral device. Files can be tested for availability as well, and can be seized, released and so on. Thus, this set of instructions is used to build models of programs within the operating system that implement the I/O reference requirements of the application program models. The first of the examples below tests for the availability of all elements in the data transfer chain.

```
IO-READY  XACTION, NOT-READY
TST-CHAN  XACTION, NOT-READY
```

The reference is to a file transaction which is the global name for the file referenced in the application group instruction. If the conditions for data transfer initiation (channel, device, controller and file ready) are not met at the simulated time of execution of this instruction, than the operating system job transaction making this test is sent to the operating system program model location NOT-READY for its next instruction.

The function of the second instruction example is similar, but with respect to the channel that connects to the global file referenced. That is, a capability exists for testing the several I/O elements individually.

There are 42 memory allocation control instructions in the SAM repertoire. The purpose of these instructions is to allocate and deallocate core memory for contiguous and non-contiguous requirements, or for both types of requirements simultaneously. If the allocation algorithm creates holes in the memory, certain of the instructions provide for management of the core memory fragments that result.

O/S Group (SAM Sample)

Program Control

PCI-CR*
RETURN
TR-PGM

| I/O Control | Allocation Control |
|---|---|
| IO-READY | EXP-CJS |
| FLSZE-TST | ALOC-NCJS |
| FILE-SZE | ALOC-CIS* |
| FILE-REL | DEAL-IS* |
| DEV-SZE | PACK |
| TEST-BUFF | HOLE-SIZE |
| TST-CHAN | SPACE |
| TST-MOUNT | |

*CIS = contiguous instruction space
IS = instruction space
PCI-CR = program control item - create

FIGURE 2

Program control instructions are used to implement the policy of the simulated system for managing inter-program relationships. They are generally employed in response to a CALL-PGM forced entry. These instructions determine the availability of a called program in the core memory, return control to a calling program, and function to manage the utilization of programs that are reentrant or recursive. For example, the instruction

TR-PGM JOB, XEMPT

sends the application transaction in the operating system model location JOB to continue its execution in the application program model XEMPT.

The Interrupt Group (Figure 3)

The interrupt instruction syntax is organized so that each interrupt can be associated with an interrupt code and a set of four interrupt operands. The interrupt code is a means for distinguishing between interrupts of the same type. For example, the interrupt codes associated with the I/O termination interrupt distinguish between the devices creating these interrupts. The interrupt operands allow data to be transmitted from the initiating source of the interrupt to the program receiving the interrupt. For instance, one of the application program instructions allows the creation of an interrupt at a later time in either the present CPU or some other specified CPU of the system. By using the interrupt operands, data given at the time of interrupt specification can be delivered to the interrupted program at the simulated moment when the interrupt occurs. The interrupt instructions are designed to employ this data in the analysis and simulation of the model's interrupt activity. It should be noted that the entrance boundary function also belongs to this group.

Interrupt Group
(SAM Sample)

Interrupt Control

ENABLE
DISABLE
GET-OPRND*
TR-XCODE*
OPR-SYS*

*OPRND = interrupt operand
XCODE = interrupt code
OPR-SYS = O/S entrance boundary

FIGURE 3

The Common Instructions (Figure 4)

There are three sub-groups of instructions designed for use in modelling both application and control software. These sub-groups contain arithmetic, transfer and queue instructions, and they also form the most direct example of operations on real data. The instructions in this group allow for addition, subtraction, multiplication, and division of integer numbers, and for sign changing and modulo operations. This means that the course of a simulation can be made to be dynamically dependent on computed data. This subject is discussed further in the next section.

It was pointed out earlier that a program model is in effect an implementation of a flowchart by means of the instruction repertoire. Such flowcharts imply conditional and unconditional transfers of control. In order that the logical representation of these flowcharts may be complete, a collection of transfer instructions is included in the group. These instructions may determine transfers that are dependent either on actual data within the simulation model or on simulated data. In the first case, conditional transfer may be based on data equality, or on the several conditions of inequality. For the second case, transfer may be based on various situations that are given probabilistic expression. For example

TR-PROB VALUE, NEXT

samples from a random distribution whose mean is 500. If the sampled value is less than the value stored in the location VALUE, then the next simulation instruction executed by the job transaction is at location NEXT within the program model. Also included in this instruction subgroup are several forms of loop control, and an end of file test that is related to simulated data transfer activities. Finally, there is an equivalent of the FORTRAN assigned GO TO, so that a program model may contain within it other program models that are executed as sub-routines.

Common Group
(SAM Sample)

| Arithmetic | Transfer | Queue |
|---|---|---|
| ADD | TR-EQ | TRQ-EMPTY |
| SUB | TR-GE | TRQ-FULL |
| MULT | TR-PROB | QMERG-TST |
| DIV | INCR-TTR | Q-RESET |
| CH-SIGN | LOOP | Q-STEP |
| | TST-EOF | QE-PLC |
| | TR-INDIR | QE-SEL |
| | | QE-MOVE |
| | | QE-FIND |

FIGURE 4

Accurate representation of the logic of a program model, particularly for a model of control software, requires that the organizational and operational logic of queues be capable of representation. The queue sub-group is designed for this purpose and is perhaps the most extensive of all of the instruction sub-groups in the repertoire. Queues are actual within SAM models and may be multidimensional. They hold real data of the simulation model and may have FIFO, LIFO or priority ordering. The queue instructions allow for the placement of entries, their retrieval, stepping through the queue, scanning the queue, and so on. In fact, all the usual operations of an extensive queue processing language are provided. For example, the instruction

```
QE-PLC  TCBQ, VALUE
```

places the value in the storage location VALUE into the queue TCBQ according to the ordering rule for that queue.

## SIMULATED TIME INSTRUCTIONS

The zero time instructions described above are intended to provide a capability for precision representation of a program flowchart. That is, the zero time instructions are used to represent the logic of what the program does, while the simulated time instructions are designed to represent the loads created by that logic. The instructions in this category therefore use up simulated time, and do this by moving the simulated time clock forward. In general, these instructions fall into three classes that are related to explicit data processing actions. These three classes of instructions are mathematical, string, and queue, and a selection from the SAM repertoire are shown in Figure 5

Simulated Time Instructions

| Mathematical | String | Queue |
|---|---|---|
| COMPUTE | EDIT | Q-COMPUTE |
| MATH | MOVE | |
| MATRIX | SCAN | |

FIGURE 5

### Mathematical Instructions

There are several types of instructions for simulating CPU load. The first of these is the COMPUTE instruction. It is used when no more precise information concerning the execution activities being simulated is available. For example, suppose that a housekeeping operating associated with program initiation is to be presented. In lieu of better information on the operations involved, it might be estimated that 20 statements are to be executed. Then when COMPUTE 20 is executed by a job transaction within a program model, the execution time for 20 statements is calculated with respect to the execution time capabilities of the CPU assigned to the job transaction.

If a statistical estimate of the number of statements to be executed is known, it may be expressed in several different probabalistic forms. For example, suppose that the number of statements whose execution is to be simulated is distributed in Gaussian form around a mean of 200 with a standard deviation of 25, from

```
COMPUTE 200, CHI, 25
```

will produce the desired, sampled simulated time.

When a mathematical calculation is known to involve a number of additions and subtractions, multiplications and divisions, the MATH instruction may be employed. For example, if a set of equations whose execution is simulated within the program model is known to contain 30 additions and subtractions, 7 multiplies and 4 divides, then

```
MATH 30, 7, 4, FLOAT
```

is used to express this computation, where the data type is specified as floating decimal.

The simulation of matrix computations is accomplished by the MATRIX instructions, where the size of the matrix is a parameter along with the data type. The inversion of a matrix of size 50 x 50 is an example:

```
MATRIX INVERT, 50, FLOAT.
```

The computational operations are to be simulated as floating decimal operations. Using this and estimates of the number of adds, multiplies and divides based on the matrix size parameter, this instruction is then executed in simulation as if it were a MATH instruction. Furthermore, the instruction assumes that the real matrix is completely contained within the real core memory. For matrices too large to fit in core, a more extensive simulation model involving data transfer activities is required.

The string processing instructions are used to represent the scanning, moving and editing of simulated data strings. The parameter specification for these instructions includes the number of data units in the simulated data in terms of characters, bytes or words. The instruction

```
STRING  SCAN, 350
```

creates an amount of simulated time equivalent to the time required to scan 350 bytes. The performance is specified as the number of time units per thousand bytes, and is part of the CPU description.

The Q-COMPUTE instruction is used to represent the load created by queue processing. Each queue in a complete model is given a symbolic, or global, name. As zero time instructions for storing, retrieving or stepping through a specified queue are executed in the course of simulation, the number of each such operation is accumulated relative to the specified queue.

For example, suppose that a queue named TCB has been defined in the operating system model. If, on a particular call on the operating system, the various logical operations as expressed by zero time activities has caused this queue to be stepped 6 times, with 6 retrievals and 5 replacements made, then the execution of

Q-COMPUTE TCB, 6, 6, 5

will cause the performance specification for stepping, retrieving and storing for this queue to be employed in determining an amount of simulated CPU time required.

## DYNAMIC DATA

If the model of a computer program is going to be able to make computations and data dependent decisions during the course of the execution of that model, then it is necessary that the data values on which such computations are based be stored in memory locations. The simplest and most obvious approach is of course to make locations available for this purpose in the host computer of the simulation. However, this would be operationally limited and linguistically stilted. A much better method is to associate actual storage locations with the elements of the simulation model itself.

### Job and Program Memory

One obvious element of the simulation model with which storage locations may be associated is the program model. In fact, we have already implied such storage locations in the several examples of the application of the SAM instruction repertoire to the creation of program models. Another, perhaps not so obvious simulation element to which storage might be attached is the job transaction. That is, we may assign to each job transaction moving through the system a set of locations for data storage, and make these locations and the data that they contain available for operation by the various simulation model instructions through which the job transaction passes.

Again, using SAM as an embodiment of the approach, we associate with each job transaction some specified number of "job value" locations. These may be thought of as traveling with the job transaction and are individually referred to by number. In SAM the syntax of the reference is JV/n where JV signifies a job value reference, and n specifies the number of the particular job value location. As an example:

ADD JV/1,JV/2,JV/3

When a job transaction executes this instruction the result of the operation is the addition of the contents of its first job value location, JV/1, to the contents of its second job value location, JV/2. The result of the addition is stored in its third job value location, JV/3. Now suppose that our program model contains a data location called ABS-VAL. Then the execution of the instruction

ABS JV/1,ABS-VAL

would operate on the number found in the first job value location of the job transaction executing this instruction, and place its absolute value in the program storage location ABS-VAL.

Thus these two memories, one of them fixed with the program model and the other moving along with the particular job transaction, may interact with one another via the instructions of the program model itself. In fact, in SAM a number of interesting programming niceties have been included such as indirect addressing. As an example, suppose that there is a location for data storage that is associated with the program model. In this location we will hold a positive integer number whose value has been computed during the course of the simulation. This numeric value will be used to designate which of the job value locations of a current job transaction is to be employed in a particular computation. Thus for example,

ABS JV/POS,ABS-VAL

will cause the absolute value of the number stored in a certain job value location to be moved to ABS-VAL. The particular job value location employed is specified indirectly by reference to the value carried in the program storage location POS. If this location has the numeric value 3 in it, then the execution of the instruction would utilize the location JV/3 of the job transaction. Of course just such computations, or any others may modify the current value of the location POS in the program model dynamically as the simulation proceeds.

## Universal Memory

In computing systems it is useful to have a set of data locations that all jobs can reference. This is also a valuable convenience in the simulation of systems of programs and can be accomplished by what we shall refer to as "universal" memory. In its simplest description this universal memory is a set of locations that are designed for storing data during the simulation so that references to these locations can be made from all program models in the simulation. In the SAM embodiment, the universal locations consist of a number of data storage positions that are referred to by ordinal number. The syntactical method for designating such a location is UV/n where UV designates the universal value reference, and n specifies the particular universal data location. When, for example,

COMPUTE UV/6

is executed by a job transaction, there will be imposed a simulated CPU load that is based on the numeric value found at the time of its execution in the sixth universal location. Indirect referencing, as illustrated above for JV locations, may also be employed with respect to UV locations. Furthermore, since data can be communicated between job transactions and the program models that they execute, and between program models and universal memory, it follows that data can be communicated between independent job transactions. This is an extremely powerful tool for the creation of precision models of systems that must synchronize the activities of independent jobs.

## Reentrant and Recursive Program Models

Since every job transaction entering the simulated system may be associated with a specified number of data locations independent of any of the program models which it may execute, it follows that the simulation system will have an immediate capability for simulating the execution of reentrant programs. To see this, suppose that each job transaction passing through a particular program model brings to that model certain real data. In the course of the simulated execution computations are made in the program model on this data. If it is then desired that this program model represent a reentrant program, we will associate all computational storage locations with JV locations of the job transaction. Now as the job transaction passes through a program model, it may encounter certain of the zero time instructions by means of which various actual computations are carried out. The results of these computations are then placed into JV locations of the job transaction. Hence, if the job transaction should execute a simulated time instruction within the program model, and if interruption should occur, then the resultant forced entry will carry the job transaction, with the currently computed actual values in its JV locations, to the operating system entrance boundary where the operating system job transaction then takes over, passing through the operating system model to carry out the simulation of all activities implied by the interrupt that has occurred. On completion, suppose that the CPU is turned to a second, different application job transaction. Evidently this new job transaction was earlier intervened from the course of its execution and is now returned to continue that execution. If, in our example, this execution is within the same program model, and this represents a reentrant program, then the events created by this second job transaction will have no effect on the computational results earlier carried out with respect to the first job transaction, since those results have moved out of the program model with this job transaction, at the time of its forced entry to the operating system.

This, of course, is the general nature of operations in a computing system with the capability for execution of reentrant programs. But what of the recursive execution programs? In this case the computing system has been organized so that a program may contain program references to itself. That is, a job may execute a CALL type of instruction in a program which calls for the execution of that same program. To generalize this, we may think of the logical sequence of program calls as forming a tree, each node of which is a program in the possible calling sequence. Then in tree structure terms, a node at a lower level may make a recursive reference to a node at a higher level. That is, some program in the calling sequence calls on an earlier program in that sequence.

For a real computing system to accomplish recursive executions it is necessary to have a storage management capability that can uniquely assign storage space for execution of a program to each different execution of that program by a particular job. Thus, if a job has entered a program for its third recursion, then there are two storage areas assigned to the first and second recursions respectively, and a new one to the third. These storage areas are furthermore usually related in a listed sequence of occurrance in which the ordering rule is last-in-first-out. That is, the recursive storage areas are treated as a push-down list.

For simulation purposes the mechanisms already discussed for providing memory associated with program models and job transactions is sufficient for realizing the needs of recursive function simulation.

Since real recursive storage areas must be associated with the real recursive program execution, then for simulation, recursive storage areas associated with the job transaction would provide the necessary storage mechanism.

In the SAM embodiment of these principles the technique has been to associate one and only one set of storage locations with the job transaction, but to assign to each execution of a program model one unique set of storage locations. That is a set of storage locations is assigned to a program model and a particular execution by a job transaction. This allocation of storage space and its association with a particular execution is invoked by the TR-PGM instruction in the SAM repertoire. When the job transaction passes through the TR-PGM function, the SAM simulator allocates the storage space and attaches that space to both the program model called and the job transaction invoking that call. Thus, if the TR-PGM references the present program model, or a program model that is already in the calling sequence group for the job transaction, a new set of storage locations for the program model which are associated with this particular job transaction will be created. These locations will be associated with the job's execution of the program model in a push-down fashion, by program model. Thus, a job may execute many program models recursively.

## AFTER WORD

The successful modeling of programs, whether application or operating systems, in the discrete event context depends very much on the language capability of the modeling system. I have gone a step further in this paper, however, by suggesting that precision in such models calls for the separation of job and program as elements of the simulation. This makes a discrete event "execution" context meaningful and gives rise to the notion of a repertoire of model instructions which are then executable.

By associating real storage locations with the model elements (job, program) I am then able to extend this approach to include the simulation of both reentrant and recursive functions.

When a capability for simulation reaches this level of representation, the simulator system becomes a microscope with which we can examine in detail any particular mote of a system's operational characteristics.

## REFERENCES

Cohen, L.J., Operating System Analysis & Design, Spartan Books, 1970; p. 52 ("CPU Transactions")

Op. Cit.; p. 56 ("the O/S Function")

Op. Cit.; p 161 ("The Prototype MPX")

Systems Analysis Machine; User's Reference Guide, Applied Data Research, Princeton, New Jersey