Proceedings of the 1989 Winter Simulation Conference
E.A. MacNair, K.J. Musselman, P. Heidelberger (eds.)

TEACHING SIMULATION WITH MOR/DS

Guy L. Curry
Bryan L. Deuermeyer
Richard M. Feldman
Texas A&M University
College Station, Texas 77843, U.S.A.

ABSTRACT

The goal in many introductory simulation courses is to focus on the concepts of simulation and minimize the time spent in learning language specific techniques. The MOR/DS (Microcomputer support for Operations Research and Management Science—Discrete Simulation) language has been developed with this goal in mind: to provide a microcomputer environment designed for teaching simulation. In this paper, we discuss its use as a teaching tool for simulation and as an introduction to languages that students will most likely use after leaving school.

1. INTRODUCTION

Simulation is an enjoyable subject to teach, because it is relatively easy to provoke interest and enjoyment in students for the topic. As instructors of an introductory course in simulation, our goals are for students to leave the course with an understanding of simulation, its practical applications and limitations, an ability to do simulation modeling, and an excitement for the topic.

Ideally, we would like to teach the basic concepts of simulation independent of any language. With a firm grasp of the concepts, students would then be able to simply learn the syntax of a particular language and, "almost instantly", be an expert. Unfortunately, it does not work that way. It is impossible to learn simulation without doing it! Therefore, if the students are to learn simulation, they must program. Thus, the next question is, "With what language?" Our goal in this tutorial is to present the use of MOR/DS as a simulation language specifically designed for teaching. Its full documentation can be found in Curry, Deuermeyer, and Feldman (1989).

2. LANGUAGE OVERVIEW

The MOR/DS simulation language was developed to provide a microcomputer environment designed for teaching simulation. The language extends the well-known block orientation of earlier languages to a full structured programming language. MOR/DS is a simulation system in which the language and the editor are integrated to provide ease of use and minimal time spent learning so that concepts instead of syntax can be stressed. On-line help, real-time animation, and user interrupt capabilities enhance learning as well as the modeling and analysis processes.

The heart of the system is a full featured text editor which provides the facilities for building and modifying models and also provides the capabilities for running simulation models. Commands (editing, file manipulation, and simulation) are accessed with a unified collection of function keys and pull down menus. The integration of the editor, compiler, and run-time components into a single system facilitates modeling, debugging and analysis. The modeler can build, debug, and execute a model without ever having to access operating system commands, or even leave the editor. This type of environment is less intimidating to beginning students and they quickly get over the fear and anxiety of dealing with a new and unknown system.

The language provides both simulation block commands (analogous to GPSS and SIMAN) and an almost complete Pascal programming structure. In this manner, routine simulation tasks are readily carried out with "blocks", while more algorithmic non-simulation tasks are directly performed with programming statements. Sufficient programming capabilities are available to support the creative needs of the modeler. The language makes use of named parameters for simulation blocks so that the meaning of a parameter is clear because the name is part of the command. However, the functionality

of MOR/DS remains sufficiently close to GPSS and SIMAN so that movement between these languages and MOR/DS is not difficult.

In the remainder of this paper, we present the MOR/DS language syntax rules, give some examples for modeling with the language, and then discuss the system's use in teaching.

3. LANGUAGE STRUCTURE

Because of the recognized advantages of structured programming languages, the design and development of MOR/DS was carried out with structured programming as a goal. An MOR/DS program has three distinct categories: definitions, control statements, and model logic. The formal structure of a program is:

```
PROGRAM
  DEFINITION
  CONTROL
  LOGIC
END.
```

The segment statements must appear in the specified order following the segment keyword identifier. Statements within a segment always end with a semicolon so that more than one statement may be on the same line, or a single statement may be written on more than one line. Case (i.e., upper- or lower-case letters) is ignored, and comments can be included almost anywhere by enclosing the text of the comment in double quotes. In this paper, we follow some typing conventions to help the reader distinguish between system words and user-supplied names. All system keywords will be typed in boldface (e.g., **Exp** and **Sqrt**), whereas user-supplied names use a typewriter font (e.g., `myVariable`).

All user-defined variables are declared and initialized in the **DEFINITION** segment. Five types of variables are supported: (1) standard mathematical types: sequences, discrete and continuous intervals, arrays, strings, functions, and reals; (2) entity types: entity pointers, attribute reals, and attribute arrays; (3) discrete simulation types: resources, queues, gates, chains; (4) statistical types: histograms, flow time and variable value statistics, and distributions; and (5) statement labels. A key feature of MOR/DS is that all data constructs are treated in the same fashion. Simulation structures such as **RESOURCES** are declared and initialized in the same manner as **ARRAYS** and other mathematical objects. MOR/DS also provides a simple mechanism for user-defined functions.

The **CONTROL** segment consists of directives to control the execution of the simulation. Simulation length controls such as time and processed entity counts, statistical reset controls, random number seed initialization, and animation lists are all located within the control segment. Another feature of the control segment is an executable logic section (**EXITLOGIC**) which is performed at the completion of each simulation run. In this logic, the control of multiple runs, statistics gathering over multiple runs, and other functions associated with the initialization and control of multiple simulations is provided. Essentially, only programming type statements, not simulation block commands can occur within the **EXITLOGIC** section of the **CONTROL** segment.

The **LOGIC** segment contains the simulation model statements. A variety of discrete simulation constructs such as entity creation and disposal, resource requests and returns, queues or waiting lists, and system as well as user-defined statistical distributions are provided. In addition, useful programming statements such as **IF/THEN/ELSE**, **WHILE**, **REPEAT/UNTIL**, **FOR**, **DOCASES**, and **PRINT** are available.

The combination of a full complement of simulation block statements and programming statements enables structured programming in the context of a block oriented language. Thus, the advantages of structured programming are combined in MOR/DS with the simplicity of the block oriented languages. A free-form named-parameter syntax for simulation blocks replaces the more common positional method used by SIMAN and GPSS. This enhances model readability and facilitates learning the syntax. An on-line help facility is available to support working with the language.

3.1. The Definition Segment

The **DEFINITION** segment contains definition and initialization of all variables used in the model development. The general definition syntax consists of an identifier, a colon followed by a type designator, an equal sign and then the variable initialization:

$$name : TYPE = initialization;$$

Name is a string of up to 15 characters followed by an optional numerical subscript (i.e., `x.5`). All variables, except for reals and functions, must have a type designator. For example, a two-dimensional array is coded in the form

```
MyData : ARRAY[<1,2>,<1,2,3>] =
    ( (11,12,13), (21,22,23) );
```

where the values within the array indicate the data entry index. Thus, `MyData[1,2]` has the value 12.

Simulation construct types, such as **RESOURCE**, are initialized through a named keyword convention. That is, instead of identifying parameter values by location within a list, the parameters are identified by a keyword. For example, a resource named `server` of capacity one is defined by

```
server : RESOURCE ={Capacity = 1};
```

Another example is a queue of capacity 10 and a FIFO queue discipline:

```
myQueue : QUEUE ={ Capacity = 10,
                   Discipline = FIFO};
```

Through the use of the named parameters, a program is easier to read than when the meaning of the parameters are simply defined by position. Because parameters are identified by name, order is not important. Furthermore, many of the parameters have default values and so most parameters do not have to be explicitly listed if the user chooses to use the default value. For example, the following definition of `myQueue` is identical to the above statement because FIFO is the default queue discipline.

```
myQueue : QUEUE ={Capacity = 10};
```

A convention followed in MOR/DS is to use braces, { }, for a list whenever order is not important; thus, a definition requiring named parameters includes the parameters in braces, whereas the values for the array use parentheses because order is important for arrays.

Other examples which illustrate the language syntax and capabilities are:

```
x = 5;
Qseq : SEQUENCE = <1,2,4..6>;
myPointer.1 : ENTITY;
f(x,y) = 14x +23y^2;
data : HISTOGRAM = { Cells = 5, MinValue
       = 0, MaxValue = 100};
MyName : QUEUE = {Discipline = FIFO,
       Histogram = data, Capacity = 3 };
```

LABELS and **ATTRIBUTES** are lists of branching labels and entity variables, respectively. For example, each entity created by the simulation has associated with it the attribute variable called `size` and the attribute array called `seq` when the following statement is used:

```
ATTRIBUTES = { size = 0, seq :
    ARRAY[<1,2,3>] = (2,5,3) };
```

3.2. The Logic Segment

The **LOGIC** segment consists of a sequence of simulation constructs (blocks) and program statements, each of which describes a particular instruction or course of action. Statements can be either simple or compound. (A compound statement is a list of statements enclosed within the keywords **begin** and **end**). Since most programming statements operate on single statements, the compound statement is the mechanism for handling multiple statements. The MOR/DS language consists of six basic statement categories:

1. simulation commands,
2. assignment statements,
3. transfer statements (goto and gosub/return),
4. conditional (if, docases),
5. repetitive (while, for, repeat/until),
6. printing (print, println),

Statement types 2–6 are patterned after the syntax of common programming languages such as Pascal and C. The simulation commands (1) are similar to the modeling statements of discrete simulation languages such as GPSS and SIMAN. The general simulation construct syntax uses a command word and a list of keyword parameter designators. Parameters not listed in a block receive the standard default values. The standard block syntax form is:

optional-label: *COMMAND* = { *Keyword* = *expression* <,*repeats*> };

To illustrate, consider a simple arrival process with a single capacity resource and a probabilistic service time. A sample fragment from a logic segment for modeling such a system would look similar to:

```
ARRIVE { Time = Expd(10) };
SEIZE { Name = server };
WAIT { Time = Cuniform(8,16) };
RELEASE { Name = server };
DEPART { };
```

Expd and **Cuniform** are the system-defined functions for random variate generation using exponential and continuous uniform random variate generation, respectively. `Server` is a resource and must

have been declared in the definition segment and could be identical to the **RESOURCE** example given in Section 3.1. The **SEIZE** block requests the resource named `server`, and the **RELEASE** block returns the resource. The resource is held for a specified length of time by the insertion of the **WAIT** statement between the **SEIZE** and **RELEASE** statements. Entities (customers) are created and scheduled into the system by the **ARRIVE** statement and the **DEPART** statement removes them from the system. Entities which cannot immediately seize the resource are automatically queued in a default resource queue in FIFO order. A named queue with a user-determined sequencing rule can be used by indicating in the **RESOURCE** declaration that a specialized queue is needed, via the **Queue** parameter. A **QUEUE** declaration statement is then used to specify the named-queue characteristics, such as the `myQueue` example in Section 3.1.

MOR/DS uses regular assignment statements (similar to BASIC and FORTRAN) rather than assignment blocks. An interesting use of the assignment statement is to store data into a **HISTOGRAM** variable. This is accomplished by merely assigning the desired value to the histogram name, such as the statement `Data = 10` where `Data` is a histogram as defined in Section 3.1. The system finds the proper data cell in the histogram and increments the cell count.

3.3 The Control Segment

The **CONTROL** segment is used to control the execution of the program, including run length, print intervals, user programmed activities after replicate runs, and animation capabilities.

The animation feature is useful for model development and is initiated through the use of the keyword **WATCHLIST**. For example, assume a queue named `myQueue` and a resource named `server` have been defined in the definition segment. The number of entities within each structure can be viewed during the execution of the model by including the statement

WATCHLIST ={ `myQueue, server` };

in the control segment. Each entity automatically has a system defined attribute variable named **WatchChar**. If **WatchChar** is blank (the default character), then when the entity joins a queue or resource, the character printed by the animation feature is a 'Q' or 'R', respectively. If, however, the **WatchChar** contains a character, then that character will be displayed during animation. The assignment of a character is accomplished in the logic segment through an assignment statement like

WatchChar = *value*;

where *value* is either a single character within quotes (e.g. `'c'`) or a numerical value which indicates which ASCII character to use.

An interrupt feature, activated by pressing the ESC key or by an entity executing an **INTERRUPT** command within the model, allows the modeler to observe various system entity lists. These lists are delineated by the WatchList setup and can include queues, resources, gates and chains. The View option of the interrupt facility allows the modeler to view various entity attributes. Other options within the interrupt feature include terminating the simulation, printing out the current statistics data, and changing the animation watch option status and delay timing.

4. EXAMPLES

Three models will be presented to illustrate points involved in teaching the simulation concepts.

Example 1. A machine important to our production process fails according to a type 4 Erlang distribution with a mean of 100 hours. We would like to simulate machine failures and generate 10 successive failure times.

```
PROGRAM  "ten failure times"
DEFINITION
  failTime = 0;
CONTROL
  StopCount = 10; "no.  failures to generate"
  Randomize =On;
LOGIC
  ARRIVE {Time = 1};
  failTime = Erlang(100,4);
  Println {'time of failure is ', failTime:7:2};
  DEPART { Quantity = 1 };
END.
```

Example 2. A machine important to our production process fails according to an unknown distribution. To estimate this distribution, we observe the following ten failure times (ordered from lowest to highest): 39.8, 41.1, 89.5, 91.5, 100.1, 124.2, 126.3, 145.5, 155.1, 181.8. We wish to use these ten data points to approximate a continuous distribution, generate 1000 failures from this approximate

distribution, and then determine the mean and standard deviation of the generated data. To do this, we will build a piecewise linear function through the *midpoints* of the data.

```
PROGRAM  "calculate mean and st.  dev."
DEFINITION
  fails : DISTRIBUTION(Continuous) =
    ( (0,39.15), (0.1,40.45), (0.2,65.3),
      (0.3,90.5), (0.4,95.8), (0.5,112.15),
      (0.6,125.25), (0.7,135.9), (0.8,150.3),
      (0.9,168.45), (1,195.15) );
  failTime = 0;   numTrails = 1000;
  total = 0;   mean = 0;
  sumSq = 0;   stDev = 0;
CONTROL
  StopCount = numTrails;
  Randomize = On;
 EXITLOGIC   "final calculations";
   mean = total / numTrails;
   stDev = Sqrt( (sumSq - numTrails*mean*mean)
      /(numTrails-1) );
   PRINTLN { 'mean and standard deviation ',
      mean:7:2, stDev:10:2 };
 END;   "end of ExitLogic"
LOGIC
  ARRIVE {Time = 1};
  failTime = fails(Random);
  total = total + failTime;
  sumSq = sumSq + failTime*failTime;
  DEPART { Quantity = 1 };
END.
```

Example 3. Consider a job shop (open 24 hours per day) that has two types of machines, Machine A and Machine B. There are also two types of jobs that come to the job shop, Type I and Type II.

Type I jobs arrive according to a Poisson process with a mean rate of 2 per hour. Each Type I job must be processed first on Machine A and then on Machine B. Each Type I job, when it leaves the system, yields an income of $500.

Type II jobs arrive in batches every hour. Twenty percent of the batches contain 3 jobs, thirty percent of the batches contain 4 jobs, and fifty percent of the batches contain 5 jobs. Type II batches are processed only on Machine A. A Type II job, when it leaves the system, yields an income of $200.

All machines act as single server systems. Processing time on Machine A is an exponentially distributed random variable with a mean time of 10 minutes (independent of job type). Processing time on Machine B is distributed according to a Weibull distribution with scale parameter 20 minutes and shape parameter 4. Any jobs that arrive when there are five jobs waiting for service for Machine A will not enter the system. If a batch of jobs arrives when there are less than five waiting, enough jobs from the batch will enter to bring the queue to capacity and the remaining jobs will not enter. There is unlimited queue capacity for Machine B.

We would like to know the mean hourly income.

```
PROGRAM  "two-machine job shop example"
DEFINITION
  queueA : QUEUE = {Capacity = 5};
  machA  : RESOURCE = {Capacity = 1,
                Queue = queueA};
  machB  : RESOURCE ={Capacity = 1};
  size   : DISTRIBUTION(Discrete) =
                ( (0.20,3), (0.5,4), (1,5) );
           LABELS = {enter, out, finish};
           ATTRIBUTES = {type = 1};
           totalCost = 0;   hourlyCost = 0;
           numHours = 100;
CONTROL
  StopTime = 60*numHours;
  WATCHLIST = { queueA, machA, machB,
                  EventChain};
  WatchStatus = SingleStep;
  EXITLOGIC   "final calculations"
   hourlyCost = totalCost / numHours;
   PRINT {'One run yields a realization of'};
   PRINTLN { hourlyCost:8:2, ' per hour' };
  END;   "end of ExitLogic"
LOGIC
     "arrivals for Type I jobs"
  ARRIVE {Time = Expd(30)};
  WatchChar = (EntityNumber MOD 26) + 97;
     "the 97th ASCII character is 'a' "
  GOTO enter;
     "arrivals for Type II jobs"
  ARRIVE { Time = 60,
           Quantity = size(Random)};
  type = 2;
  WatchChar= (EntityNumber MOD 26) + 65;
     "the 65th ASCII character is 'A' "
enter:
  SEIZE {Name = machA, Qexcess = out};
  WAIT {Time = Expd(10)};
  RELEASE {Name = machA};
  IF type = 2 THEN
    begin
      totalCost = totalCost + 200;
      DEPART { };
    end;
  SEIZE {Name = machB};
```

```
  WAIT {Time = Weibull(20,4)};
  RELEASE {Name = machB};
  totalCost = totalCost + 500;
  DEPART { };
out: DEPART { };
END.
```

5. USE OF MOR/DS IN CLASS

There are three key issues that students should gain an understanding of during their first exposure to simulation: (1) the statistical appreciation of processes to be simulated, (2) the entity driven nature of discrete simulations together with the use of future event chains, and (3) the statistical appreciation of output analysis. We shall indicate the use of MOR/DS in helping the students gain an appreciation for each of these issues. We would like to emphasize at this point that we are simply providing some observations regarding successful approaches in introducing simulation to students. There are many alternative approaches, many of which may be better than what we have observed, but we present our observations in the hope that some instructors may find them beneficial.

5.1. Statistical Variation of Processes

The first two examples in the above section serve to emphasis two of the principles involved in dealing with randomness. To begin the process of learning to appreciate the generation of random numbers, several small examples similar to Example 1 can be given to students. The syntax needed to program examples like these is simple enough so that students can do their own programming and experimentation after one day of exposure to MOR/DS. When doing examples like this in class, if the statement **Randomize = On** is used, the class can compare output and immediately see variation.

Example 2 can be used to illustrate two major sources of error common to modeling processes. Most of the time, the modeler does not know the exact distribution(s) of interest; therefore the first source of error is the sampling error when obtaining the relevant probability laws. The second source of error is the statistical variation inherent in a simulation. Of course, if the sampling error is significant, then no matter how long the run and how many replicates are made, the model will be inaccurate. This point is easily emphasized to the student by combining the first two examples. Example 2 indicates there is an unknown distribution generating failure times, and it gives a data set of failure times. Instead of using the given data set, have the students generate their own data through the use of Example 1. That is, use the output from Example 1 as the random sample for Example 2. Thus, the student can see exactly where the error is occurring. If the program in Example 2 is run for different lengths, further evidence is produced emphasizing errors generated with statistical experiments.

5.2. The Future Event Chain

Upon first encounter, discrete simulation programs (models) are difficult to comprehend, because while they look similar to regular computer programs they do not behave, in general, like regular programs. Consequently, until the basic operation of block-oriented simulation systems is understood, there is an element of mystery that may cause the student difficulty with building models.

Let us begin with a brief and simplified discussion of the operation of a standard computer program. Internal to the underlying system is an instruction counter that indicates the next instruction to execute. After each instruction is processed, this instruction counter is incremented. The cycle of "get instruction, execute instruction, and increment counter" is repeated until some termination criterion is met. The instructions that are executed are usually calls into a library of run-time routines established by the compiler; these eventually lead to low level instructions understood by the computer (microprocessor) itself. In fact, the instruction counter is maintained by the microprocessor.

MOR/DS makes use of this same conceptual structure except for two critical differences: each entity maintains its own instruction counter, and only active entities can cause the execution of a command (that is a program statement or a simulation block). Furthermore, MOR/DS provides its own controller, and calls to its event-routine library perform the necessary commands.

MOR/DS maintains both active and inactive entities. While the external behavior suggests that several entities are active concurrently (this is relative to simulated time), the fact is that exactly one entity is actually active (this is relative to real time). It is the operation of the controller, through the active entity and the inactive entities, that gives a simulation program its distinctive characteristics.

The inactive entities are stored on either a special list called *The Future Event Chain* or lists such as queue lists. Each entity on the Future Event

Chain has a time at which it is scheduled to become active, and this time may be established in a number of ways. The two most obvious methods are by an **ARRIVE** block or by a **WAIT** block, each of which calculates the scheduled time by adding a user-specified value to the current simulation time. Of course, other less straight forward mechanisms can do this as well. When an event is scheduled, the time is stored in an internal entity attribute and the entity is placed onto the Future Event Chain. The members of Future Event Chain are sorted by increasing time and then by FIFO; the entity with the earliest scheduled time is at the front of this list.

At any point during the operation of the simulation, the active entity is executing commands while the Future Event Chain keeps track of other entities and when they are scheduled to occur. At some point, the active entity encounters a delay command (which means an event time is calculated and the entity is stored in the Future Event Chain), a blocking command (as in an attempt to seize a resource and the entity is placed on a queue list), or a depart command. The controller then takes the front element from the Future Event Chain, and this entity becomes the active entity and begins executing the commands as indicated by the entity's internal command counter. So long as no simulation clock time is consumed several different entities may go through the inactive — active — inactive cycle, thus giving the impression of concurrency.

The **WATCHLIST** and **View** option are helpful features in tracing the logic flow of a model for students just beginning to understand the concept of entity-driven programs. We demonstrate this through Example 3. Notice in Example 3 that the command **WatchStatus = SingleStep** is used. This command forces the simulation to pause after each step until a key is pressed. Another command you should notice in the **LOGIC** segment is the two assignment statements for **WatchChar**. These two statements indicate that as soon as an entity enters the system, they will be represented in the animation screen as a lower case letter for Type I jobs and as upper case letters for Type II jobs. When the program is executed the following will appear on the screen.

```
- - - - - - - - - - - - - - - - - - -
Watch Simulation          Time = 0.00

    QUEUEA      |
    MACHA       |
Resource Queue  |
    MACHB       |
EVENTCHAIN      |CCCCC
- - - - - - - - - - - - - - - - - - -
```

At time zero, the only entities in the system are those created initially by the **ARRIVE** blocks. These entities appear as 'C' because their **WatchChar** is blank and thus they use the default character for a chain. The five entities appearing in the Future Event Chain refer to the single entity created by the first **ARRIVE** block and the batch of (randomly generated) four entities created by the second **ARRIVE**. We now hit the space bar and the following appears.

```
- - - - - - - - - - - - - - - - - - -
Watch Simulation          Time = 10.50

    QUEUEA      |
    MACHA       |b
Resource Queue  |
    MACHB       |
EVENTCHAIN      |bCCCCC
- - - - - - - - - - - - - - - - - - -
```

From this we see that the first entity has seized resource **MachA** and has entered the **WAIT** block. That first entity is also on the Future Event Chain because the action of the **WAIT** block is to place an entity on the Future Event Chain.

To view the individual entities, the ESC key is pressed which causes the following to appear.

```
System clock =  10.50

Entity:     1 is at:[ 8]     WAIT

  C - Continue
  P - Print current results
  W - Watch status            ON
  D - Delay duration 300
  S - Single-Step  ON
  V - View lists
  Q - To terminate run
  A - To abort system
```

Since we wish to view lists, the V key is pressed; thus producing:

```
View Entities                Clock Time = 10.50
Num  Sym   Time    Key  |    Attribute Values
- - - - - - - - - - - - - - - - - - - - - - - -
    MACHA
  1  b  18.720   0.000 |    1.000
EVENTCHAIN
  1  b  18.720   0.000 |    1.000
  6     51.769   0.000 |    1.000
```

```
  2      60.000   0.000 |   1.000
  3      60.000   0.000 |   1.000
  4      60.000   0.000 |   1.000
  5      60.000   0.000 |   1.000
- - - - - - - - -
    Press any key to continue and ESC to quit
```

From this view list, we see that the current simulated clock time is 10.50 and the single entity in the system is in **MachA** and is scheduled to release that machine at time 18.720. There is another entity scheduled to enter the system at time 51.769 and a batch of four entities scheduled to enter the system at time 60. We next press the space bar to continue, which produces the interrupt menu. We press C to continue which produces the next animation screen as follows

```
- - - - - - - - - - - - - - - - - - -
Watch Simulation            Time = 18.72

    QUEUEA      |
    MACHA       |
Resource Queue  |
    MACHB       |b
EVENTCHAIN      |CbCCCC
- - - - - - - - - - - - - - - - - - -
```

Notice that the future event chain continually reorders its entities so that they appear in increasing order of event time. We see from this that the next event to happen will be the arrival of another job. Every time we hit the space bar, the watch list will advance one event. If we hit the space bar two times the screen will appear as follows:

```
- - - - - - - - - - - - - - - - - - -
Watch Simulation            Time = 60.00

    QUEUEA      |DEF
    MACHA       |C
Resource Queue  |
    MACHB       |g
EVENTCHAIN      |gCCCCCC
- - - - - - - - - - - - - - - - - - -
```

We next hit the ESC key and then the V key to view the entities:

```
View Entities                 Clock Time = 60.00
Num  Sym   Time    Key  |     Attribute Values
- - - - - - - - - - - - - - - - - - - - - - - -
      QUEUEA
  3   D  60.000  0.000 |   2.00
  4   E  60.000  0.000 |   2.00
  5   F  60.000  0.000 |   2.00
      MACHA
  2   C  74.276  0.000 |   2.00
      MACHB
  6   g  70.145  0.000 |   1.00
EVENTCHAIN
  6   g  70.145  0.000 |   1.000
  2   C  74.276  0.000 |   2.000
  8      120.000  0.000 |   1.000
  9      120.000  0.000 |   1.000
 10      120.000  0.000 |   1.000
 11      120.000  0.000 |   1.000
  7      175.082  0.000 |   1.00
- - - - - - - - - -
    Press any key to continue and ESC to quit
```

By using a computer in the classroom and discussing these screens, students will begin to understand the concept of the entity-driven nature of discrete simulation faster than mere words can convey the concept.

5.3. Analysis of Output

It is important to begin emphasizing to students as soon as possible that simulations are statistical experiments, and one data point is not enough. A subsegment of the **CONTROL** segment is called **EXITLOGIC**, which is designed specifically to help in the analysis of multiple runs or obtaining periodic information within one run. At the end of a simulation run, or if an entity enters a **HALT** statement, the **EXITLOGIC** will be executed. There are three special statements designed for control purposes. These are (1) the **CONTINUE** statement, which resumes the run at the current **ClockTime** when the **EXITLOGIC** is finished, (2) the **RESET** statement, which resumes the run at the current **ClockTime** but clears all statistics variables, and (3) the **CLEAR** statement, which clears all entities, sets the **ClockTime** to zero, clears all statistics variables, and then resumes the run.

Through the use of **EXITLOGIC**, it becomes possible to do the necessary calculations to obtain results from replicate runs, batch means, regenerative procedures, or other special purpose procedures for output analysis. An example of its use for batch means in estimating hourly revenue from an M/M/1/5 queueing system (each customer yields $5 revenue) follows:

PROGRAM "EXITLOGIC for batch means"
DEFINITION
que : **QUEUE** = {Capacity = 4};
ser : **RESOURCE** = {Queue = que};
batchSize = 10; "hours within batch"
numBatch = 25; "number of batches"

```
  batchCount = 0;
  cost = 0;
  sumCost = 0;  ssCost = 0;
  mean = 0;  std = 0;
  LABELS = { out };
CONTROL
  BlockListing = Off;
  Randomize = On;
 EXITLOGIC  "batch calculations";
   batchCount = batchCount + 1;
   sumCost = sumCost + cost;
   ssCost = ssCost + cost*cost;
   IF batchCount < numBatch THEN
     CONTINUE
    ELSE  begin
     mean = sumCost/numBatch;
     std = ssCost - numBatch*mean*mean;
     std = Sqrt( std/(numBatch-1) );
     PRINT {'mean and standard deviation:'};
     PRINTLN {mean:7:2, std:10:2 };
    end;  "end of ELSE"
 END;   "end of ExitLogic"
LOGIC
  ARRIVE {Time = Expd(5)};
  SEIZE {Name = ser, Qexcess = out};
  WAIT {Time = Expd(5)};
  cost = cost + 5;
  RELEASE {Name = ser};
out:
  DEPART { };
      "arrivals for batching"
  ARRIVE {Time = 60*batchSize};
  cost = cost / batchSize;
  HALT;
  cost = 0;
  DEPART { };
END.
```

As is illustrated in the above program, most programming statements are available within the **EXITLOGIC**. The utilization of this subsegment can facilitate output analysis and help in proper analysis.

REFERENCE

Curry, G. L., Deuermeyer, B. L., and Feldman R. M. (1989). *Discrete Simulation: Fundamentals and Microcomputer Support.* Holden-Day, Inc., Oakland, California.

AUTHORS' BIOGRAPHIES

GUY L. CURRY is a Professor in the Departments of Industrial Engineering and Agricultural Engineering. He received the B.S. and M.S. degrees in mathematics from the University of Oklahoma and Wichita State University, respectively, and a Ph.D. in industrial engineering from the University of Arkansas. Prior to joining Texas A&M University, he was an operations research analyst with the Boeing Company and a senior operations research analyst with Sun Oil Company. Dr. Curry has received two Texas A&M University system awards for research, and while at Texas A&M, he has been principal or co-principal investigator on over two million dollars of externally funded research. His research interests include systems analysis, mathematical optimization, and modeling of biological systems. He is a member of ORSA, TIMS, IIE, ASAE, and is a registered Professional Engineer in Texas.

Guy L. Curry
Industrial Engineering Department and
Agricultural Engineering Department
Texas A&M University
College Station, TX 77843, U.S.A.
(409) 845-5576

BRYAN L. DEUERMEYER is a Professor of Industrial Engineering. He received a B.A. in chemistry from the University of Minnesota, and an M.S. and Ph.D. in industrial engineering from Northwestern University. He served on the faculty of the Krannert Graduate School of Management at Purdue University before joining the faculty at Texas A&M University. Dr. Deuermeyer is an associate editor of *Management Science.* His research interests are in computer science related to modeling and analysis, mathematical optimization, and combinatorics. He is a member of IIE, ORSA, TIMS, Sigma Xi, and is a registered Professional Engineer in Texas.

Bryan L. Deuermeyer
Industrial Engineering Department
Texas A&M University
College Station, TX 77843, U.S.A.
(409) 845-5049

RICHARD M. FELDMAN is a Professor of Industrial Engineering. He received the B.A. and

M.S. degrees in mathematics from Hope College and Michigan State University, respectively. He also received an M.S. in industrial engineering from Ohio University and a Ph.D. in industrial engineering from Northwestern University. Prior to joining Texas A&M University, he spent three years as an operations research analyst with the Goodyear Atomic Corporations and one year as an operations research analyst at the Michael Reese Medical Center. Dr. Feldman has received several departmental and college awards for teaching while at Texas A&M. His research interests are in stochastic processes, simulation, and mathematical and biological modeling. He is a member of IIE, ORSA, TIMS, and is a registered Professional Engineer in Texas.

Richard M. Feldman
Industrial Engineering Department
Texas A&M University
College Station, TX 77843, U.S.A.
(409) 845-5585