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

PRODUCTIVITY TOOLS IN SIMULATION: SIMSCRIPT II.5 AND SIMGRAPHICS

Otis F. Bryan, Jr.
CACI Products Company
1600 Wilson Boulevard, Suite 1300
Arlington, VA 22209, U.S.A.

ABSTRACT

The SIMSCRIPT II.5 simulation language with its integrated dynamic graphics package, SIMGRAPHICS, substantially reduces the time and effort to produce a simulation when compared to general purpose languages.

Its structured English syntax improves readability and reduces some of the need for documentation. This syntax is supported by powerful libraries to relieve the programmer of substantial amounts of work by pushing it onto the computer.

SIMGRAPHICS adds graphics to a program to display results dynamically. Graphics in most programs takes thousands of lines of code. SIMGRAPHICS reduces this task to a few lines of code with mouse-and-menu graphics editors.

All of this adds up to major improvements in productivity in programming. Rapid prototypes with dynamic graphics can be written in a few days. Code is open to inspection by non-programmers. Results are faster and more reliable.

INTRODUCTION

Discrete event simulations comprise a substantial part of the programming world. They range from simple models in academia to massive programs in government.

General purpose programming languages are inherently inefficient in the simulation world. They lack to constructs necessary to handle routine matters such as scheduling, filing and dynamic graphics.

To solve these problems, programmers write thousands of lines of code to handle simple bookkeeping tasks before they can ever get to the modeling questions. This is worse with graphics because they have to use the primitive commands of systems such as GKS.

The SIMSCRIPT II.5 programming language provides the tools for discrete event simulation as part of its libraries. This frees the the modeler to get into his main task -- building the model.

This tutorial describes the main features of the language and illustrates them with a simple example.

SIMSCRIPT's CONCEPT

Simulation involves the passage of time in the life of some object. For example, passengers arrive at an airport, wait for a passenger agent, get a boarding pass and leave.

General purpose programming languages, such as Pascal, C and FORTRAN, lack the constructs to make time pass in a simulation. The programmer has to write these constructs before he can get onto writing the simulation.

In general, SIMSCRIPT's simulation constructs are built around the concept of a process. This is a routine that describes what happens to the object as it moves through time. For example, here is the SIMSCRIPT code that processes a passenger in a terminal.

```
Process PASSENGER

    Define ARRIVAL.TIME as a real variable

    Let ARRIVAL.TIME = time.v
    Request 1 PASSENGER.AGENT(1)
    Let WAITING.TIME = time.v - ARRIVAL.TIME
    Wait exponential.f (MEAN.SERVICE.TIME, 1)
        minutes
    Relinquish 1 PASSENGER.AGENT(1)

End "PASSENGER
```

Before getting into the details, a comment about style is in order. SIMSCRIPT II.5, unlike many other languages, is not case sensitive: ARRIVAL.TIME and arrival.time are the same variable.

While SIMSCRIPT will not punish you for failing to capitalize a word, it does require that variables be spelled correctly. If you misspell a word, you will get a warning message: Local variable used only once. This is part of SIMSCRIPT's substantial error checking capability.

As a matter of style all SIMSCRIPT words are shown as capitals and lower case; user defined words are all capitals. This way the substance of the model is apparent as you scan the code.

The two apostrophes at the end of the process routine are the beginning of a comment.

The first statement declares ARRIVAL.TIME to be a local, real variable.

The PASSENGER.AGENT is a resource. The PASSENGER has to have a PASSENGER.AGENT before he can get a boarding pass. So he requests one. If one is available, the next line of code is executed immediately.

If a PASSENGER.AGENT is not available, control passes to the timing routine. The timing routine files this PASSENGER in a queue waiting for a PASSENGER.AGENT and starts execution of the next process.

When a PASSENGER.AGENT becomes available and this PASSENGER is first in the queue, the timing routine removes the PASSENGER from the queue and schedules him to continue execution.

When this PASSENGER is the next process to be executed, control returns to the process routine at the line after the request statement.

The next statement calculates the amount of time the PASSENGER had to wait for a PASSENGER.AGENT. Time.v is the current simulated time.

The Wait statement represents the passage of time while the PASSENGER gets his boarding pass. The amount of time is drawn from an exponential distribution with a mean of MEAN.SERVICE.TIME using a random number stream 1.

As with the resource, control passes back to the timing routine. The timing routine files the PASSENGER in a queue of pending events, called an event set. This queue is ranked by the time of reactivation.

Reactivation time for this PASSANGER is the current simulated time plus the amount of time drawn from the exponential distribution.

When this PASSENGER is next up for execution, control returns to the statement after the wait statement. The PASSENGER relinquishes the PASSENGER.AGENT so that someone else can use it and disappears from the scene.

The great advantage of the construct is the the modeler can write the steps of the process in structured English. Having done this, he only has to create passengers when he wants them, and SIMSCRIPT will handle all the details of scheduling and execution.

Notice that the process is clear about what is supposed to happen. This reduces the chances of logical errors in coding. More importantly, an airline employee can read the code and know whether it represents what "really" happens.

ENTITIES, ATTRIBUTES AND SETS

Behind the scene, SIMSCRIPT does quite a bit of bookkeeping. This relieves that modeler of that tedium.

A PASSENGER is represented by a process notice. A process notice is a temporary entity in SIMSCRIPT. Temporary entities are similar to records in Pascal and structures in C. The PASSENGER.AGENT is a permanent entity and is represented by an array.

Both entities have attributes or characteristics. The PASSENGER has nine attributes including time.a, the reactivation time.

The term, set, refers to a doubly linked list. When the timing routine puts the PASSENGER in the event set, it files a temporary entity or record in a doubly linked list ranked by time.a or reactivation time.

Processes are executed in order of occurrence in the event set. To execute the next process, the timing routine removes the first process notice from the event set and sets time.v to the reactivation time from the process notice. It then passes control back to the process routine.

When control passes back to the timing routine, it will do one of three things with the process notice.

- If it is at a request statement, the notice is filed in a queue waiting for a resource.

- If it is at a wait statement, the notice is filed in the event set according to reactivation time.

- If it is at the end of the process, the notice is destroyed by returning its memory back to the memory manager.

MEMORY MANAGEMENT

SIMSCRIPT uses dynamic memory allocation for most of its constructs.

Anything requiring more than one or two computer words gets memory to store it from the memory manager. This includes string variables, arrays and entities.

The address to this memory is stored in a pointer variable with the same name as the data structure. For example when a process notice for a PASSENGER is created, the address of the first word of its block of memory is stored in the pointer variable, PASSENGER.

While it is a little confusing to the user to have a structure and variable with the same name, it is unambiguous to the program. This improves readability by letter the user refer to the PASSENGER without have to use two separate names.

Dynamic memory allocation lets small computers such as PC's run powerful programs. Even major programs exceeding 150,000 lines of SIMSCRIPT II.5 code can run on work stations.

CREATING INSTANCES OF PROCESSES

The process routine shows what happens to one passenger. We need to create many passengers to run a simulation. The general technique involves creating a second process that functions as a passenger generator.

```
Process PASSENGER.GENERATOR

    Define I as an integer variable

    For I = 1 to 120 do

        Activate a PASSENGER now
        Wait exponential.f
            (MEAN.INTERARRIVAL.TIME, 2) minutes

    Loop "I = 1 to 120

End "PASSENGER.GENERATOR
```

PASSENGERs are created one at a time with some time passing between creation of each PASSENGER. In actuality, this is the wait between arrivals that normally occur at an airport: generally passengers do not show up all at once.

The PASSENGER.GENERATOR will create 120 passengers. An alternative to the For statement is:

```
        Until time.v >= RUN.TIME
```

In this case PASSENGERs would be created until some simulated time had passed, say eight hours.

The Activate statement creates the process notice for this instance of the passenger, sets the reactivation time, time.a, to time.v (i.e. now) and files it in the event set.

The PASSENGER.GENERATOR then waits some amount of simulated time before looping and creating the next PASSENGER. It passes control back to the timing routine. The timing routine sets the PASSENGER.GENERATOR's reactivation time to the current time plus the time drawn from this exponential distribution.

It then files the PASSENGER.GENERATOR in the event set, gets the next process notice and starts executing it.

When the PASSENGER.GENERATOR is first in the event set, the timing routine returns to the statement after the Wait statement, loops, creates the next passenger and goes back on the event set.

RUNNING THE SIMULATION

All that remains is to start the simulation and build some infrastructure to support this model.

```
Main

        Call READ.DATA
        Activate a PASSENGER.GENERATOR now
        Start simulation

End "Main
```

Every SIMSCRIPT program must have a Main routine. This is where execution starts.

In this case Main calls a subroutine (READ.DATA), creates the first instance of a process with an activate statement and passes control to the timing routine with the Start Simulation statement.

When the simulation finishes, control returns to Main. Here the simulation ends immediately.

There must be at least one process notice in the event set before starting the simulation. Anytime the timing routine finds the event set empty, it assumes the simulation is finished and returns control to the routine with Start Simulation.

If you forget to activate a process before starting the simulation, you will tie the world's record for the shortest running simulation in history.

INPUT AND OUTPUT

Input and output can be done with text or graphics. We'll look at a textual method first.

Because the program uses data from the user, we'll add a subroutine to get the data from the user.

```
Routine READ.DATA

    Print 2 lines with time.v thus

**.***    Enter the mean service time in minutes:

    Read MEAN.SERVICE.TIME

    Print 2 lines thus

        Enter the mean interarrival time in minutes:


    Read MEAN.INTERARRIVAL.TIME

End "READ.DATA
```

The Print statements will write something to the current output unit, namely the screen. What is written is on the two lines following the Print statement.

First there will be a blank line, because it is one of the two "format" lines. The next line will start with the current value of time.v in place of the asterisks followed by whatever is written on the remainder of the line.

This is a what-you-see-is-what-you-get print statement. There is no counting of characters, columns etc. Just type in what you want to see, and the print statement will put it there.

Similarly the Read statement does not require counting of characters. It just reads the next field from the current input unit and sets MEAN.SERVICE.TIME to that value.

A field is any string of characters between two blanks or a blank and a carriage return. Consequently the data values can go anywhere in a file, just as long as they come in the correct order. That is, the value for MEAN.SERVICE.TIME has to come before the value for MEAN.INTERARRIVAL.TIME.

THE PREAMBLE

Variables in SIMSCRIPT are either local or global; there is no intermediate state. SIMSCRIPT routines are recursive, and memory for a local variable is created for each call to that routine. The value in that variable is visible only to that call of that routine.

At the other end of the spectrum, global variables are visible from anywhere in the program.

A SIMSCRIPT program generally begins with a preamble where all global variables must be defined.

```
Preamble

    Define MEAN.SERVICE.TIME,
        MEAN.INTERARRIVAL.TIME and
            WAITING.TIME
    as real variables

    Define LOW,
        HIGH,
        DELTA
    as integer variables

    Processes include
        PASSENGER.GENERATOR and
        PASSENGER

    Tally WAITING.TIME.HISTOGRAM (LOW to HIGH
        by DELTA) as the histogram of WAITING.TIME

End "Preamble
```

The two Define statements declare the list of variables as global real or integer variables.

WAITING.TIME is an output variable whose value is established in the process, PASSENGER.

All processes have to be declared in the preamble.

MONITORING AND AUTOMATIC STATISTICS COLLECTION

SIMSCRIPT has a feature, called monitoring, that automatically intercepts control of the program and passes it to a subroutine. When the subroutine is finished control is passed back to the statement where it was intercepted and the program proceeds.

The Tally statement in the preamble creates a one dimensional array to store the data for the histogram.

Its size is determined by the variables LOW, HIGH, and DELTA. If these values are 0, 100 and 5 respectively, there will be 21 cells in the one dimensional array.

The Tally statement declares WAITING.TIME to be a variable monitored on the left. That is, every time WAITING.TIME in the process, PASSENGER, is about to change, control is passed to a library routine that updates the histogram.

This one declaration frees the programmer from having to put statements throughout the program to update the histogram. It eliminates the possibility that he will forget to do it some place and get erroneous results.

This method can also be used to collect statistics for means, variances, maxima, minima and counts.

PRESENTATION GRAPHICS

Output and input can be done with the Print and Read statements shown above. This is useful in dealing with precise data and files.

There are times when the data needs to be displayed graphically, and SIMSCRIPT automates that process.

In this example, we will use the dynamic graphics capability of SIMSCRIPT to display the histogram. There are three steps:

- Use the SIMGRAPHICS editor to lay out a histogram on the screen. The editor has a number of standard presentation graphics including graphs, histograms, clocks and meters. Using a mouse-and-menu system, we change the attributes, such as color, line style and location.
- When the screen layout is finished, its characteristics are saved in a file (e.g. WAIT.GRF) in the program's directory.
- Then it is attached to the program with two statements.

The Tally statement is modified in the preamble and a second statement is added to the Main routine.

```
Tally WAITING.TIME.HISTOGRAM (LOW to HIGH
      by DELTA) as the dynamic histogram of
      WAITING.TIME.
```

The word, dynamic, causes the histogram on the screen to redraw itself every time WAITING.TIME.HISTOGRAM changes; that is, every time WAITING.TIME changes.

In the Main routine prior to Start Simulation, add:

```
Display WAITING.TIME.HISTOGRAM with
      "WAIT.GRF"
```

This statement tells SIMSCRIPT to use the image in WAIT.GRF to display the data in WAITING.TIME.HISTOGRAM.

From here on SIMSCRIPT will take care of the details.

ANIMATION

Dynamic graphics is not limited to presentation graphics. SIMSCRIPT has a similar capability for animation.

Suppose we have a process that simulates an airplane flying from one point to another. It is a straight forward matter to make the image of an airplane fly across the screen.

```
Preamble

    Processes include AIRPLANE

    Dynamic graphic entities include AIRPLANE

End "Preamble
```

Animated movement requires the use of processes, because time has to pass.

The second statement adds attributes to the process notice to cover movement. This includes attributes for location, velocity and image. In addition it brings up the library routines that change location and update the image automatically.

```
Main

    Activate an AIRPLANE now
    Display AIRPLANE with "AIRCRAFT.ICN"
    Start simulation

End "Main
```

The image of the airplane with be drawn with a mouse using a SIMGRAPHICS editor. When done, the data representing it will be saved in a file called AIRCRAFT.ICN.

As with the presentation graphics, the image of the aircraft is attached to the process with the display statement. Every time the AIRPLANE changes its location, SIMSCRIPT will erase the old image and redraw it at the new location.

```
Process AIRPLANE

    Let location.a(AIRPLANE) = location.f(200, 300)
    Let velocity.a(AIRPLANE) = velocity.f(520, pi.c/4)

    Wait 1.5 hours

End "AIRPLANE
```

The initial location is set with the location.a statement. This is normally the only time the user changes the location directly. From now on SIMSCRIPT will change the location automatically.

The two parameters, 200 and 300, are the x and y coordinates of the AIRPLANE. These could be variables or expressions.

The velocity vector is set with the velocity.a statement. The first parameter is speed (i.e. 520 knots). The second parameter is direction in radians (i.e. 45 degrees from north). The velocity.f function changes these parameters to Cartesian coordinates for purposes of calculation.

Given the initial location and velocity vector all that remains is to figure out how long to fly in that direction and to do so. The Wait statement says to fly for 1.5 hours. When done, the AIRPLANE will be 780 miles northeast of where it started.

Reduced to bare essentials, animation in SIMSCRIPT involves controlling the velocity vector and waiting time. SIMSCRIPT takes care of the rest of it behind the scenes.

FORMS EDITORS

Graphics input is the third feature of SIMGRAPHICS. Rather than use a text editor, it is possible to build a form and attach that to a program.

Whenever data is needed, the form is automatically displayed.

To enter data the user clicks on the appropriate data box. A question mark appears, and the user types in the value. When all data is entered, the user clicks on a button and the simulation proceeds.

Building a form works the same way presentation graphics and animation does: design the form with a mouse-and-menu editor, save it in the directory and attach it to the program.

Attaching a form to the program is a little more complicated than attaching a graph. A form can have multiple data boxes and buttons.

When the user selects a box, the program has to know what to do with the data from the box. Each box has an identifier. When the user clicks on the box, SIMSCRIPT not only gets the value, but it records the identifier of the box and passes control to a routine written by the user.

This routine is essentially a large case statement. Each case is the identifier of a particular box. Under each case are the statements as to what to do when the box is chosen.

For example, instead of typing in data in the Passenger Agent Problem, suppose we used a form. The user routine would look like this

```
Routine CONTROL

    Given
        FIELD.ID,
        FORM
    Yielding
        STATUS

    Select case FIELD.ID

        Case "SERVICE"

            Let MEAN.SERVICE.TIME
                = ddval.a(dfield.f("SERVICE", FORM))

        Case "ARRIVAL"

            Let MEAN.INTERARRIVAL.TIME

                = ddval.a(dfield.f("ARRIVAL", FORM))


        Case "RUN"

    Endselect "FIELD.ID

End "CONTROL
```

The form has two data boxes and one button. (Technically, the form has three fields).

By clicking on the data box for the mean time between arrivals and entering a new value, the user has selectED the field with the identifier, ARRIVAL. This is passed to this routine through the input parameter, FIELD.ID.

When control is passed to this routine, it executes the statements under the case ARRIVAL. Here it reads the value in the field, "ARRIVAL", from the form whose name is stored in FORM.

Then it returns control to SIMSCRIPT which waits for the user to select another field.

When the user clicks on the button, RUN, the form disappears and the simulation starts.

The forms editor has a number of other fields the user can employ in his form. These include scroll boxes, menu bars, radio buttons, and icon lists. This material is too extensive to pursue here. The reader should refer to the SIMGRAPHICS manual, (1988), for details.

SIMANIMATION AND SIMGRAPHICS

The developers of SIMSCRIPT and related programs have tried to develop a language that will save time and effort for the users. This led to the SIMSGRAPHICS editors with their mouse-and-menu systems.

At the same time it was recognized that these could not do everything the user wanted to do. So primitive commands have always been available to the "do-it-yourself" user.

SIMGRAPHICS refers to the three editors for presentation graphics, animation and forms along with their supporting libraries.

SIMANIMATION refers to the primitive graphics commands that are an integral part of the SIMSCRIPT II.5 programming language. These commands were based on the concepts of the Graphical Kernel System (GKS). Because general GKS libraries are too slow for simulation, the statements have been implemented with SIMSCRIPT libraries.

For more details see the SIMGRAPHICS User's Guide and Casebook (1988).

SIMGRAPHICS has recently undergone a major revision. SIMGRAPHICS II integrates the editors into a single editor with a common method of developing screen layouts for presentation graphics, animation and forms.

SIMGRAPHICS II has been designed to work with X-Windows. As X-Window implementations become available on various systems, the entire language including graphics can be ported quickly to new machines.

GENERAL PURPOSE PROGRAMMING

SIMSCRIPT is a powerful general purpose language in its own right, independently of its uses as a simulation language.

It has general purpose constructs, such as assignments, branches, loops, inputs, outputs and subroutine calls. Its data structures include scalars, arrays, records and lists.

For example, suppose the user wanted to create an airline with a fleet of 50 airplanes. Here is the code to do it:

```
Preamble

    Permanent entities

        Every AIRLINE has
```

```
            a NAME and
            owns a FLEET

                Define NAME as a text variable

    Temporary entities

        Every AIRPLANE has
            a TAIL.NUMBER and
            a GROSS.TAKEOFF.WEIGHT and
            belongs to a FLEET

            Define TAIL.NUMBER as an integer
                variable
            Define GROSS.TAKEOFF.WEIGHT
                as a real variable

End "Preamble
```

Two entities have been declared.

- The AIRLINE is a permanent entity and will be stored as an array. It has one attribute, NAME.
- The AIRPLANE is a temporary entity and will be stored as an individual record. It has two attributes, TAIL.NUMBER and GROSS.TAKEOFF.WEIGHT.

The FLEET ISowned by the AIRLINE and containing AIRPLANEs at members is a set. Technically it is doubly linked list.

```
Main

    Define I as an integer variable

    "Create ten airlines and read their names from a file


    Create every AIRLINE(10)
    For each AIRLINE do
        Read NAME(AIRLINE)
    Loop "each AIRLINE

    "For AIRLINE number 7, create 50 AIRPLANEs, read
    their"attributes and put them in the FLEET of
    that AIRLINE

    Let AIRLINE = 7
    For I = 1 to 50 do
        Create an AIRPLANE
        Read TAIL.NUMBER(AIRPLANE),
            GROSS.TAKEOFF.WEIGHT(AIRPLANE)

        File the AIRPLANE in the FLEET(AIRLINE)

    Loop I = 1 to 50 do

End "Main
```

The entity, AIRPLANE, is a two-dimensional array. The "rows" are the attributes, and the "columns" are the individual airlines. In addition to the NAME attributes, it has two pointer attributes containing the addresses of the first and last AIRPLANEs currently in the set, FLEET.

The For statement loops through each AIRLINE from 1 to 10. Inside the loop it reads the names of the airlines from the current input unit, one field at a time.

AIRLINE is the name of the entity and the name of a variable used as an index for the array.

The statement, For I = 1 to 50 do, sets up a loop to create AIRPLANEs one at a time.

Creating an airplane gets the memory to hold the data for a single AIRPLANE. Its address is stored in a pointer variable, AIRPLANE, used to refer to this entity.

After creating an AIRPLANE, values for the two attributes are read from the current input file using a the free format READ statement.

The file statement adds the AIRPLANE to the set FLEET. Technically, this involves modifying pointers in the doubly linked list, FLEET. Each AIRPLANE has two pointer attributes for the addresses of predecessor and successor of this AIRPLANE in the FLEET.

It is possible to loop through the fleet looking for a particular AIRPLANE and doing something with it. For example,

```
    For each AIRPLANE in the FLEET(AIRLINE)
        with TAIL.NUMBER(AIRPLANE) = "N627D"

    Find the first case

    If found

        Print 1 line with TAIL.NUMBER(AIRPLANE)
            thus
        Tail number: ***********

        Remove the AIRPLANE from the
            FLEET(AIRLINE)
        Destroy this AIRPLANE

    Endif "found
```

SIMSCRIPT goes through the FLEET looking for the AIRPLANE with the TAIL.NUMBER equal to N627D. If it finds it, SIMCRIPT drops out of the loop and goes to the If found statement.

Since SIMCRIPT found the AIRPLANE it was looking for, it will print the TAIL.NUMBER, remove the AIRPLANE from the FLEET and release the AIRPLANE's memory back to the memory manager.

If the AIRPLANE were not in the FLEET, it would finish the loop with AIRPLANE pointing to the last entity in the set. Since we may not want to remove and destroy this one, the If found statement, being false, will prevent that.

This is a simple example of the power of SIMSCRIPT as a general purpose programming language.

There is certainly far more to SIMSCRIPT than this. Anyone interested in more details should contact the author.

CONCLUSION

This short tutorial has gives some sense of the savings in using SIMSCRIPT in both simulation and general purpose programming.

As a rule of thumb it takes about four times as much FORTRAN code to do the same thing as a SIMSCRIPT program. Using other general purpose languages not only takes more work, but it increases the chances of undetected errors.

CACI provides a number of services in addition to selling compilers under a 60 day free trial. Training courses, consulting and program development are available. For more information, contact CACI Products Company at 703-875-2919.

REFERENCES

----, SIMGRAHPICS User's Guide and Casebook, CACI, La Jolla, California, 1988.

----, SIMSCRIPT II.5 Programming Language, CACI, Los Angeles , California, 1987

Russell, Edward C., Building Simulation Models with SIMSCRIPT II.5, Los Angeles, California, 1983.

SKIP BRYAN manages the Simulation and Modeling Department for CACI Products Company. He received a B.S. in Mechanical Engineering from MIT, an M.S. in R & D Management from the University of Southern California and an MBA from the University of Chicago. His current interests include specification and design of large scale simulation models.