A GPSS-LIKE LANGUAGE IN THE SAS SYSTEM FOR
DISCRETE EVENT SIMULATION

Gretchen K. Jones, National Center for Health Statistics
Michael A. Greene, The American University

ABSTRACT

This paper describes a program written in the SAS system which allows constructing a simulation using a sub-set of the GPSS language. Using this program, a discrete event model can be encoded as data (following the CARDS statement) and then run in the SAS environment. The advantage of this approach is the ability to use the statistical and graphics procedures in SAS for monitoring of the simulation during the run and for post simulation analysis.

1. INTRODUCTION

The purpose of this paper is to describe a prototype of a GPSS discrete event simulation language compiler implemented in the SAS system. In its present form, the compiler reads a subset of GPSS blocks and statements as if they were data. These blocks and statements contain the model, i.e., the code which represents the movement of transactions through a queuing network. The compiler parses the input data to check for errors and then sets up the necessary internal data structures. Transactions are then generated and sequenced through a set of operations to run the simulation. At the conclusion of the run an event trace dataset can be produced. This file can then be immediately processed by various SAS statistical or graphical procedures or saved for later analysis.

GPSS is an abbreviation for General Purpose Simulation System which was developed by Gordon and Lawton at IBM during the 1950s. There are currently a variety of different implementations of the language for many different machines. See Gordon (1975), Schriber (1974), Norden (1971), and Henricksen and Crain, (1983). Other simulation languages include SIMSCRIPT (Russell, 1980) and MICRO PASSIM (Barnett, 1982), hosted in Pascal.

The unusual aspect of this work is that the prototype language compiler is written in the language provided in a statistics package. The principal reasons for selecting SAS software were (1) the completeness of the language in the data step, and (2) the immediate access to high quality statistics and graphics procedures.

Section 2 of this paper briefly addresses language related issues. Section 3 contains a description of the subset of GPSS implemented in SAS and some details of the construction of the compiler. Section 4 shows an example of output from a simulation run including graphics. Section 5 describes future directions for this work.

2. LANGUAGE ISSUES

There are a large number of issues to be confronted by the designer of a simulation system. In many ways, the issues are no more and no less complex than the design of any programming language.

Why a simulation language at all? The argument for a special purpose language as compared with writing a simulation in a general purpose language centers on convenience in programming and modeling. Using a simulation language presumably reduces the cost of software development (see Henricksen, 1983). As an example of the contrast between using a general purpose language and a simulation language, figures 1 and 2 show programs for simulating an M/M/k queue (exponentially spaced arrivals, exponentially spaced service times and k servers; see Gross and Harris, 1974 for details) in FORTRAN and GPSS. The FORTRAN program is adapted from Brately, Fox and Schrage (1987). While only the main program is shown in figure 1 and the complete program in figure 2, some major differences in writing in each language are apparent. The principal difference is that in FORTRAN the code must be written to correctly sequence each event in the order in which it occurs while in GPSS it is sufficient to describe the process which occurs for a single entity moving through the queue.

Figure 1
Main Program for A Queuing Simulation in FORTRAN

```fortran
PROGRAM QUE
COMMON/TIMETAY/ TIME
INTEGER ARRIVAL, ENDSRV, QUEUE(O), SRFREE/O/,
EVENT, STORE
INTEGER ASSRV/1234589/, SCRWR/19933984/
REAL TIME, NTTARG, X
PARAMETER (ARRIVAL=1, ENDSRV = 2, STORE = 3)
EXTERNAL NTTARG, INIT, PUT, GET, SITMARK
NOST = 0
ENSSRV = 0
CALL INIT ; SET TIME = 0, EMPTY QUEUE
X = ZETMARK(O) ; LIMIT THE ARRIVAL EVENT
C
C SEND THE FIRST ARRIVAL
CALL PUT (ARRIVAL, NTTARG(ASSRV))
C AND THE END OF THE SIM
CALL PUT (STOREM, 480)
C
C MAIN LOOP
99 CALL GET (EVENT)
IF (EVENT .EQ. ARRIVAL) THEN
NOST = NOST + 1
CALL PUT (ARRIVAL, NTTARG(ASSRV))
IF (SRFREE .GT. 0) THEN
SRFRHE = SRFREB - 1
ENSSRV = ENSSRV + 1
CALL PUT (ENSSRV, TIME + 1)
1 NTTARG(SRFRHE)
ELSE
QUEUE = QUEUE + 1
ADD CUST TO QUEUE
END IF
GO TO 99
ELSE IF (EVENT .EQ. ENDSRV) THEN
IF (QUEUE .LT. 0) THEN
ENSSRV = ENSSRV + 1
QUEUE = QUEUE - 1
CALL PUT (ENSSRV, TIME + 1)
1 NTTARG(SRFRHE)
ELSE
SFRERE = SRFREB + 1
END IF
GO TO 99
C ELSE
C PRINT THE STATISTICS & FINISH
END IF
END
```

This is only the main program and there are no statements for output. The complete program in GPSS together with a FORTRAN function for logarithms is
shown in figure 2 below.

**Figure 2**
Simulation Program in GPSS/H with supplementary FORTRAN function

<table>
<thead>
<tr>
<th>Block or Statement</th>
<th>OPERANDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORAGE</td>
<td>Capacity</td>
</tr>
<tr>
<td>GENERATE</td>
<td>1=uniform</td>
</tr>
<tr>
<td></td>
<td>Mean</td>
</tr>
<tr>
<td></td>
<td>Half width</td>
</tr>
<tr>
<td></td>
<td>2=exponential</td>
</tr>
<tr>
<td></td>
<td>Rate</td>
</tr>
<tr>
<td></td>
<td>3=normal</td>
</tr>
<tr>
<td></td>
<td>Mean</td>
</tr>
<tr>
<td></td>
<td>SD</td>
</tr>
<tr>
<td>ENTER</td>
<td>Storage Label</td>
</tr>
<tr>
<td>ADVANCE</td>
<td>1=uniform</td>
</tr>
<tr>
<td></td>
<td>Mean</td>
</tr>
<tr>
<td></td>
<td>Half width</td>
</tr>
<tr>
<td></td>
<td>2=exponential</td>
</tr>
<tr>
<td></td>
<td>Rate</td>
</tr>
<tr>
<td></td>
<td>3=normal</td>
</tr>
<tr>
<td></td>
<td>Mean</td>
</tr>
<tr>
<td></td>
<td>SD</td>
</tr>
<tr>
<td>LEAVE</td>
<td>Storage Label</td>
</tr>
<tr>
<td>TRANSFER</td>
<td>Label of Block</td>
</tr>
<tr>
<td>ASSIGN</td>
<td>Parameter Value</td>
</tr>
<tr>
<td></td>
<td>1=replace</td>
</tr>
<tr>
<td></td>
<td>2=Add</td>
</tr>
<tr>
<td></td>
<td>3=Subtract</td>
</tr>
<tr>
<td>TERMINATE</td>
<td>Decrement</td>
</tr>
<tr>
<td>START</td>
<td>Value of Termination</td>
</tr>
<tr>
<td>END</td>
<td>0=Output Trace</td>
</tr>
<tr>
<td></td>
<td>1=No Trace</td>
</tr>
</tbody>
</table>

**Prototype GPSS Language Features**

The prototype contains a partial implementation of 3 GPSS statements and 7 GPSS blocks. These are shown in Figure 3. The statements, which are considered control structures in GPSS are STORAGE, START and END. The remainder are blocks and are used to describe the model. Both blocks and statements have operands (constants placed to the right of the block or statement names) which are used to control the flow of execution or to establish model conditions.

Access to high quality graphics. It is difficult to imagine conducting any sort of statistical analysis without associated graphics. There are no substitutes for stem and leaf plots or scatter plots to detect outliers in bivariate relationships. For trivariate relationships, scatter plots using different plot characters to represent the third dimension, two-dimensional contour plots and three-dimensional projections can be useful. Additionally, graphics can be helpful in debugging by providing means for organizing and displaying an event trace.

Why SAS software and Why GPSS? SAS has high quality statistics and graphics routines; also the availability of a general purpose language in the data step. GPSS is well known in the simulation community, is in the public domain and its relatively rigid syntax facilitated developing the compiler.

3. DESCRIPTION OF THE PROGRAM

The next section describes the features of the GPSS language which are implemented in our prototype. It is followed by an example. The last part of this section contains some details of the SAS language used in the implementation.

![Diagram showing GPSS language features in the prototype](image-url)
Example. Figure 4 denotes how the model is coded to simulate an M/M/1 queue.

Figure 4
Simulation Program for a M/M/1 Queue

<table>
<thead>
<tr>
<th>Time</th>
<th>Number</th>
<th>Code</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000485</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0.000485</td>
<td>1</td>
<td>2</td>
<td>7047</td>
</tr>
<tr>
<td>0.052630</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0.124534</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0.171947</td>
<td>1</td>
<td>3</td>
<td>7047</td>
</tr>
<tr>
<td>0.171947</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>0.246155</td>
<td>3</td>
<td>3</td>
<td>7047</td>
</tr>
<tr>
<td>0.246155</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>0.290813</td>
<td>4</td>
<td>3</td>
<td>7047</td>
</tr>
<tr>
<td>0.246155</td>
<td>4</td>
<td>2</td>
<td>7047</td>
</tr>
<tr>
<td>0.290813</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>0.333348</td>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0.333348</td>
<td>5</td>
<td>2</td>
<td>7047</td>
</tr>
<tr>
<td>0.352883</td>
<td>6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0.350820</td>
<td>5</td>
<td>3</td>
<td>7047</td>
</tr>
<tr>
<td>0.360820</td>
<td>6</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Compiler Description. The compiler is written as a data step in about 550 lines of SAS software. There are three main phases, (1) input (2) clock update and (3) scan phase. The input phase reads in the program (as data following CARDS), checks for syntax errors, sets up the internal arrays and schedules the very first event from each GENERATE block in the model. These events go on the Future Events Chain (see Schriiber, 1974 for a description of this GPSS feature). The next two phases are executed sequentially until the end of the program. The second phase, clock update, advances the simulation clock to the time of the next scheduled event as found on the Future Events chain and moves all events scheduled to occur at that time from Future Events to the Current Events chain. The third phase, scan, activates all the events on Current Events chain by moving each transaction as far as it can go. The main data structures of the program are 6 explicitly subscripted arrays of which 5 are logically two dimensional. FEC is the future events chain and CEC is the current events chain. TAI contains information on each active transaction. The Clock Update Phase is shown in figure 5 below.

Figure 5
SAS Code Fragment for the Clock Update Phase

```
CLOSE:
IF NEXT = 0 THEN DO;
   PUT 'SCHEDULE EMPTY';
   STOP;
END;
IF TAI(FEC(NEXT)) = CLOCK THEN GO TO SETEVNT;
IF TAI(FEC(NEXT)) < CLOCK THEN DO;
   PUT 'CLOCK RUNNING BACKWARDS';
   STOP;
END;
CLOCK = TAI(FEC(NEXT)); // ADVANCE CLOCK TO TIME OF NEXT EVENT;
SETEVNT:
EVENT = FEC(NEXT); // EVENT = NEXT TRANSACTION ON FEC */
NEXT = NEXT - 1;
J = 1;
ASK:
IF CEC(1,J) = 0 THEN GO TO SETEC;
J = J + 1;
GO TO ASK;
SETEC:
CEC(1,J) = EVENT; // PUT EVENT ON CURRENT EVENTS CHAIN */
IF NEXT = 0 THEN GO TO SCAN;
IF TAI(FEC(NEXT)) = CLOCK THEN GO TO SETEVNT;
ELSE GO TO SCAN;
```

Event Traces. Event traces are useful for debugging models but usually provide too much information for statistical analysis. Examples of the event trace are shown below.

A typical example of the first 17 records from event trace output dataset is shown in figure 6. These records were created with the program in figure 4 for the M/M/1 queue with the rates of the exponential distributions in the B operand at 9.5 for the GENERATE block (arrivals) and 7.5 for the ADVANCE block (services). Transaction 1 is shown to have been generated (Code 1) at 0.000485 time units on the first line of the figure and immediately to have entered service (Code 2) at storage 7047 on the second line of the figure. The next event was the creation of transaction 3 at time 0.05263 on the third line and then transaction 4 at time 0.12453. Line 5 shows that transaction 1 completed service (Code 3) at 0.171947 and then was terminated (Code 4). With the storage free at 0.171947, transaction 3 then was moved in on the next line. The remaining lines describe the activities associated with transactions 4 and 5. Transaction 6 is shown to arrive at 0.352883 but it had not yet started service as of the last line of figure 4.

Figure 6
First 5 Transactions

Statistical and Graphical Programming. Figure 7 shows a SAS program for processing the event trace. It contains 3 data steps and 5 procedures. In the first data step (Data Out1), the system size variable (SysSize) is created following the Select statement by adding 1 for each arrival and subtracting 1 for each service completion. Transaction number 2, the timing transaction, is removed from the dataset. Also, the records from event types 2 (start service) and 4 (terminate) are deleted because these events are not of interest to the present analysis. In keeping with an interest in the start-up behavior of this system, transactions later than number 400 were deleted.

4. SAMPLE OUTPUT

Sample output from the program includes event traces, statistics and graphical output. These are described below.
SAS Code for Post Processing the Event Trace

Data Out1;
    Retain SysSize 0 LastTime 0;
    Set Out;
    * Get event trace;
    If Trano = 2 or EvType = 2 Then Return;
    If Trano > 400 Then Delete;
    Select (EvType);
    When (I) SysSize:=SysSize+ 1;
        * Arrival;
    When (3) SysSize:=SysSize-1;
        * Departure;
    otherwise; End;
    Output;
Proc Sort Data=Out1; By Trano;
Data Out2;
    Retain TimeIn 0 Sys;
    Set 0ut1;
    Keep Trano Sys Timeln TimeOut Wait Lwait;
    If EvType=1 then Do; Timeln=Time; End;
    If EvType = 3 Then Do; TimeOut=; Time; Wait:=TimeOut-TimeIn; LWait=Lag(Wait);
    Output; End;
Proc Univariate Freq Plot Data=Out2; Var Sys;
Proc Corr Data=Out2; Var Wait Lwait Sys Trano;
Data 0Ut2; Set OUt2; Century = Int(Trano/100)+~;
Proc G3d Gout=M;y.Cat; Scatter Trano "* Sys ;; Wal.t;
    By Century;

Statistical and Graphic Output. In Figure 8, output from Proc Univariate for the System size variable is shown. The average system size encountered by the first 100 observations is shown to be 4.85 customers. The mode of system size is 1, (that is the arriving transaction itself) as shown also in the stem and leaf plot, occurring for 18 of the 100 transactions, equaling to a 18 per cent probability that the server was idle when the transactions arrived. The maximum system size was 13 which was seen by only 1 transaction. The distribution of system size is shown by the quantiles and the stem and leaf plot.

Figure 8
Selected Output from Proc Univariate

<table>
<thead>
<tr>
<th>VARIABLE</th>
<th>MEAN</th>
<th>N</th>
<th>SE</th>
<th>VAR</th>
<th>CV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sys</td>
<td>4.85</td>
<td>485</td>
<td>3.29</td>
<td>10.83</td>
<td>10.83</td>
</tr>
<tr>
<td>Min</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Median</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N50</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N75</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N95</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N99</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A bubble plot showing the relationship between the waiting time and system size for each transaction in the first 100 is shown in figure 10. The radius of the bubble is proportional to the system size.

Figure 10
Bubble Plot of Time in the System vs Transaction Number
5. FUTURE DIRECTIONS

The experiments conducted with SAS software reveal that system to be a useful language for a simulation compiler. These routines give the user the ability to tailor the output analysis to the research problem and to generate statistically valid results.

Future plans for the development of the compiler include enhancing the structures for the flow of control. This would mean extending the options available with the TRANSFER block (especially subroutine mode) and implementing some version of the TEST block. This would probably be done differently than as in GPSS, more along the lines of the If...Then...Else construct. Another important extension are the GPSS User Chains which allow the user to manage a queuing process. Installing this feature would require the SPLIT block (as copies of transactions are usually found on User Chains) and the LINE and UNLINK Blocks. The last extension would allow the equivalent of GPSS/E asservariables (similar to savevalues) for storage and accumulation of data during the run. Then once the compiler has been stabilized, the next logical step would be to rewrite it in PL/I.

The current version of the compiler is available on electronic mail on a test basis free of charge. It may be obtained by contacting M. GREENE @ AUVM on Bitnet.

REFERENCES


Norden Division of United Technologies (1971), GPSS/Norden Simulation Language, Norwalk, CT.