Proceedings of the 1991 Winter Simulation Conference
Barry L. Nelson, W. David Kelton, Gordon M. Clark (eds.)

SIGMA TUTORIAL

Lee Schruben

School of O.R.I.E.
Cornell University
Ithaca, NY 14853

ABSTRACT

SIGMA (denoting Simulation Graphical Modeling and Analysis) is an interactive graphics approach to discrete event simulation. This tutorial contains a brief introduction to simulation graph modeling with SIGMA. In addition, there is a discussion of some recent advances in the SIGMA software and an example. Among the recent enhancements to SIGMA are graphs for output analysis, ranked lists, and a facility for creating an English description of the simulation graph.

1 BASIC APPROACH AND NOTATION

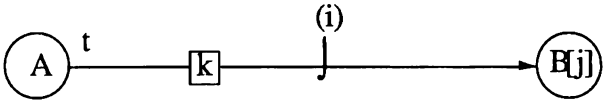
SIGMA was developed for teaching discrete event simulation on personal computers. Although SIGMA is very easy to learn, it is completely general and can be used to create large-scale simulations that run faster than many commercial codes. Indeed, any computer program can be created with SIGMA [1]. Outside the classroom, SIGMA has been used successfully to study such diverse systems as banks, food production facilities, and computer architectures as well as in military and manufacturing applications.

In October of 1990, The Scientific Press published SIGMA and began distributing it to academic institutions [2]. Despite a simple user interface, SIGMA is a fully functional simulation, modeling, and analysis system. For best results, an IBM PS/2 class system under MS-DOS 5.0 is recommended; however, people have reported running SIGMA successfully on Apple, SUN, and DEC equipment. SIGMA has been implemented under WINDOWS 3.

In SIGMA, stochastic discrete event systems are represented as simple dynamic graphs. Models are created by drawing these graphs on a computer screen with a mouse. A few mouse clicks will run the simulation. By means of its graphic displays, students learn how various events in the system interact. SIGMA provides a "logical" animation of the model. Each run produces output charts and traces on a disk that are compatible with most spreadsheet and statistical analysis packages.

In SIGMA, the relationships between system events are represented by a simple directed graph based on the familiar "event graph" concept [3]. Event graphs are an elementary way of capturing the dynamics of discrete event systems.

An event graph is a directed graph that organizes the objects of a simulation into a model. Pictorially the vertices of the graph represent state changes that are associated with the various events in the simulation. The edges of the graph represent the logical and temporal relationships between the vertices. For example, suppose the following is part of a simulation graph,



This edge is interpreted as follows:

> Whenever event A occurs, if condition (i) is true, then event B will be scheduled to occur t time units later with parameter(s), j, having the values(s) of expression(s) k.

Once students understand this single network symbol, they have learned about simulation graphs. SIGMA is the computer implementation of these graphs. Several recent textbooks use such graphs to explain simulation models [4,5,6]. While SIGMA nicely complements these textbooks, it might also be used to introduce more complicated commercial simulation languages. A SIGMA graph with GPSS(SLAM,SIMAN)-like vertex names is a good way to introduce process oriented models and helps make the corresponding models in GPSS(SLAM,SIMAN) much more plausible.

Persons familiar with process network modeling might think of SIGMA as having a single, "generic," model building block as opposed to many special-purpose blocks found in most commercial network-oriented languages. In fact, the primary motivation for developing event graphs was to combine the advantages of network modeling with the more general and efficient but abstract event-oriented simulation approach without losing the ability to develop intuitive process models.

2 SPECIFIC FEATURES OF SIGMA

Some specific features of **SIGMA** are highlighted below.

WRITE_In_ENGLISH: Exploiting the graph structure of SIGMA, English translations of the model can be generated with a few mouse clicks. The "Write in English" command automatically translates the graphical model into English. The English model description is an excellent tool for finding errors in model logic. If the English translation does not make sense, it is highly unlikely that the program will function correctly. It is much easier to recognize nonsense in an English sentence than in a complex computer program. To illustrate: the following is an exact SIGMA-generated English description of a "start service" event in a simple queueing system graph. The code for this event is given later.

> The START event models the start of service.
> This event causes the following state change(s):
> SERVER=0
> QUEUE=QUEUE-1.
> After every occurrence of the START event:
> Unconditionally, service will take 5 minutes; therefore, schedule the LEAVE event to occur in 5 time units.

The English translation can be used for model verification, explaining a model to others, and as an alternative or supplement to time-consuming physical animations. This is also valuable in grading homeworks, comparing what students claim their models are doing with what SIGMA says it is doing.

OUTPUT CHARTS: By clicking on the "Output Chart" command button, several charts are created that allow one to visually analyze output files. Students can explore various initial truncation, batching, and run duration strategies and quickly see the results. For those so inclined, sufficient statistics are also given for inference and interval estimation. However, the graphs alone will meet most analysis needs.

The following charts can be selected:

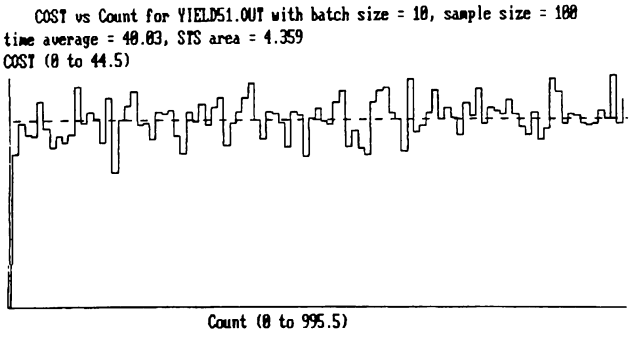


Figure 1. Line plots, which show how variables change over time.

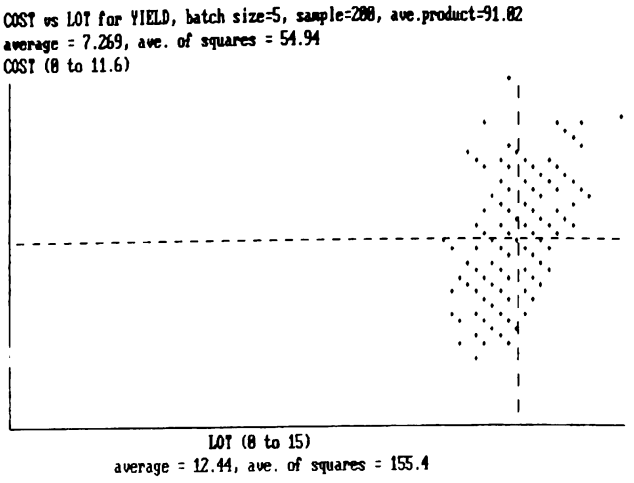


Figure 2. Scatter plots, which show the relationship between pairs of variables

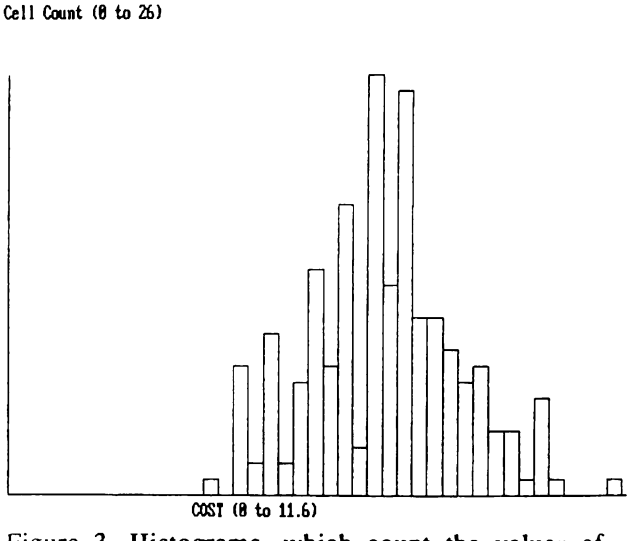


Figure 3. Histograms, which count the values of variables

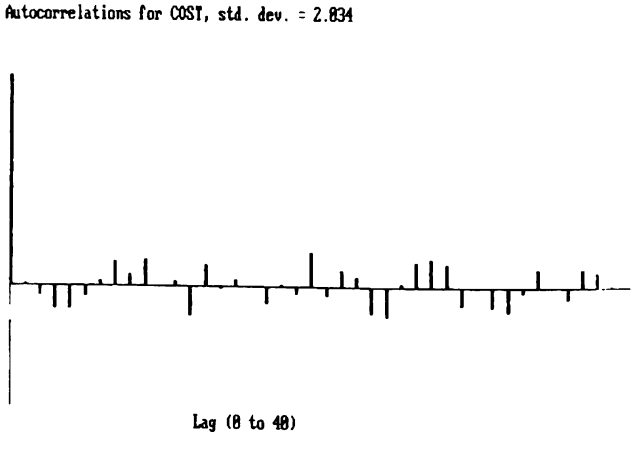


Figure 4. Autocorrelation plots, which show serial dependence in the output

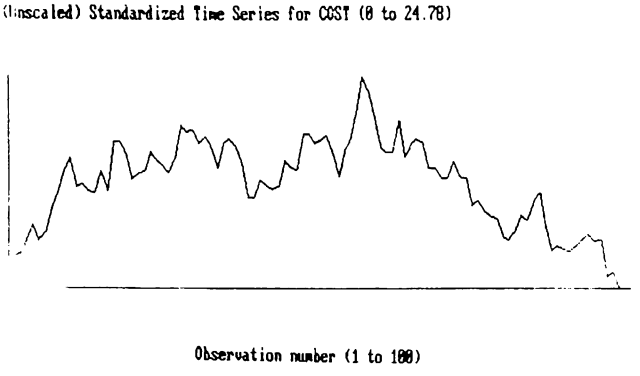


Figure 5. Standardized Time Series (STS), which can be used to detect trends

The standardized time series plot should appear to be symmetric about zero if there is no trend in the data. A trend might be due to model warm-up bias and inadequate initial truncation. This plot will be pulled above or below the zero line by initialization bias.

Note that Figure 5 is for the same data as Figure 1. A subtle trend in the data can now be readily detected.

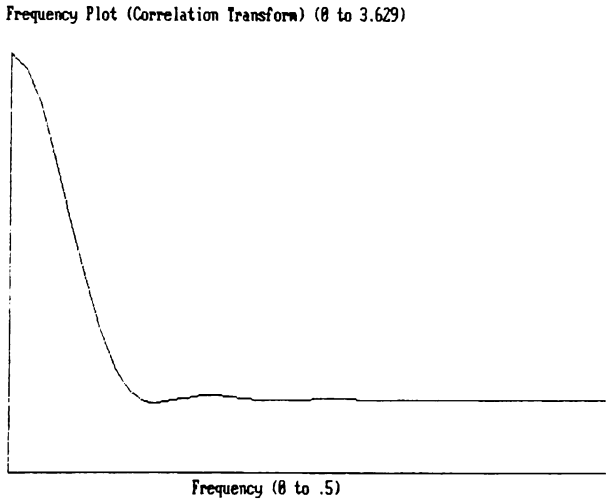


Figure 6. Frequency Spectra, which can be used to detect cycles

USER DEFINED FUNCTIONS: A USER type variable is actually an executable function on the default disk drive, written perhaps in FORTRAN, C, Pascal, or BASIC. As long as a simple calling convention is followed, these executable files do not even have to be linked with SIGMA. USER variables can be used in expressions like any other variable. Extensive use of user functions requires a workstation or a PC running with MS-DOS 5.0.

INSTANT INPUT VERIFICATION: There is instant spelling, syntax, and computation checking of all expressions. Unlike mere syntax checkers, SIGMA verifies that each computation is legal by actually executing expressions when they are entered.

AUTOMATIC SOURCE CODE GENERATION: If a compiled program is desired, a SIGMA graph will automatically generate portable standard C or Pascal simulation source code. The generated code is extensively commented so that no prior knowledge of C or Pascal is necessary. Furthermore, unlike most conventional simulation languages, complete source code for the entire SIGMA-generated simulation is provided; there are no proprietary "mystery" functions. Since these features make the modeling process easier to understand, more class time can be spent on other crucial elements of the simulation process such as validation, experimentation, analysis, and communication of the results.Students immediately see what is going on since their SIGMA expressions and descriptions are embedded (in capital letters) right in the code. To illustrate, the following is the SIGMA-generated C code for the "start service" event referred to earlier in this paper. Some non-essential comments and code were removed to save space, but nothing was added.

```
/* START OF SERVICE */
int START
 {
 /* state changes */
   SERVER=0
   QUEUE=QUEUE-1.

 /* schedule future events */
 /* SERVICE TAKES 5 MINUTES */
   event_time = current_time + 5;
   event_type = 4;
   event_priority = 5;
   schedule_event();
 }
```

Compare this with the SIGMA-generated English description given earlier for this event to see how the vertex and edge descriptions tie things together. The Pascal source code generator is similar.

SIGMA has been used as a C or Pascal program generator for applications that have nothing to do with discrete event simulations. The graph becomes an executable flow-chart enriched to permit blocks of code (vertices) to be easily executed in a state-dependent, dynamically-changing sequence. Thus, programs are conceptually written in a plane rather than as a linear stream of code with sometimes confusing branching. Loops look like loops and all branching (if-then-else, call, goto) is identical.

MODELING RESOURCE CONSTRAINTS: If several types of resources are needed to schedule an event, simply list the names of these resources on the scheduling edge in a SIGMA graph.

USING SEVERAL RANDOM NUMBER STREAMS: Function definitions in C make it very easy to make your SIGMA random number stream a "vector" of random number streams.

SIMPLE MANAGEMENT OF RANKED QUEUES:

SIGMA now supports easy management of priority ranked queues. Ranked queues occur whenever the order of service might differ from the order of customer arrival. Management of priority queues is done with the simple PUT and GET functions.

Attributes of individual transient entities (customers in a queueing system) can be assigned to the elements in the array ENT[]. For example, ENT[0] might be the customer arrival time, ENT[1] the class of service, and ENT[2] the amount of product to be purchased. The state change vector

ENT[0]=CLK, ENT[1]=CLASS, ENT[2]=DEMAND

might model the relevant attributes of a customer. The ENT[] array is used exclusively as a buffer for customers joining ranked queues using the PUT{} and GET{} functions. The array, RNK[LINE], determines a customer's position in the LINE.

The pair of SIGMA functions, PUT{} and GET{}, allow customers (with their various attributes) to be put into and retrieved from ranked queues.

The PUT{OPTION;LIST} function, places the current contents of the ENT[] array in the LIST. LIST is a number, variable, or function that identifies the queue to be joined. The OPTIONs include:

- FIF (first-is-first) inserts the new entity after the last record on the LIST.
- LIF (last-is-first) inserts the new entity before the first record on the LIST.
- INC (increasing) the LIST is ranked by increasing values of ENT[X], where X=RNK[LIST] is the ranking entity attribute.
- DEC (decreasing) the LIST is ranked by decreasing values of ENT[X], where X=RNK[LIST] is the ranking entity attribute.
- EVN (even) when the values of ENT[0] for two entities are even, then the tie is broken by increasing values of ENT[2] with remaining ties broken by "first-is-first."

GET{OPTION;LIST) removes a record from the specified LIST places its contents in the ENT[] array. The OPTIONs include:

- FST (first) removes the first element of LIST.
- LST (last) removes the last element of LIST.

For example: either pair, (PUT{FIF;1} and GET{FST;1}) or (PUT{LIF;1} and GET{LST;1}), could be used in modeling a first-come-first-served queue. Both PUT{} and GET{} return 1 if successful and 0 otherwise. They are typically used as an edge condition where 0=FALSE. Naturally, these functions should not be used as part of an edge condition that might be false, they are executed regardless. If used in a state change, PUT and GET should appear on the right-hand side of an equation, such as

QUEUE[N] = QUEUE[N] + PUT{FIF;N}.

This state change will increase QUEUE[N] (the length of the nth queue) by 1 when the customer with attributes currently in the ENT[] array is put into this queue. The customer can later be removed (with attributes placed in ENT[]) with the state change,

QUEUE[N] = QUEUE[N] - GET{FST;N}.

DISTINCT EXPERIMENTAL FRAME: With the new SET function, students can run complete experimental designs within a single graph.

3 AN EXAMPLE

The simulation of a series of limited-space queues is a very common application. In this example, a series of N tandem queues are modeled with different service times and different buffers (waiting spaces for work in progress). A machine cannot start a new job until it passes its current job down the line. If there is no space in front of a queue, then the upstream machine becomes "blocked."

The event graph for this system is given in Figure 7 and the exact SIGMA-generated English description is given in the Appendix. This model is pretty straightforward except perhaps for the DONE event vertex. The DONE vertex models the event where the jth machine is done with a job. The status of this machine might change to idle (1) or to blocked (-1), depending on whether or not there is room to unload the job into the downstream queue.

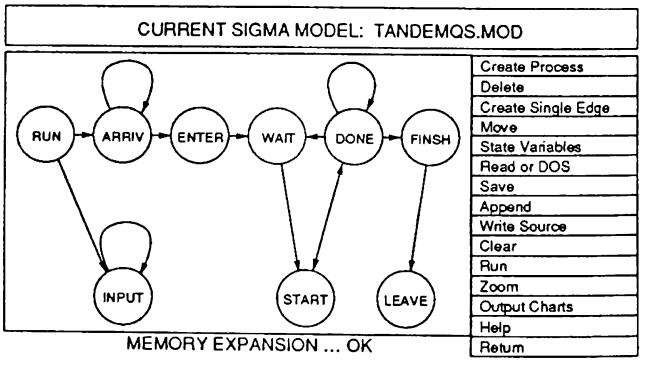


Figure 7. SIGMA graph for example

If the machine is not blocked and can unload its job, then it might unblock the upstream machine (change status of the j-1st machine from -1 to 1). The unblocking of machine j-1 might also unblock machine j-2, and on up the line. The system temporarily behaves like a "pull" production system. All the logic for changing the status of servers involves simply passing the value of the status of the jth server into the DONE vertex as an attribute of its scheduling edge.

In a process model of this system (where each job is identified), the simulation will run slower and use more memory if many jobs enter the system. Furthermore, a

process-oriented simulation which distinctly represents each individual machine (perhaps for the sole purpose of animation) would be limited to a relatively small number of machines.

In the event-oriented model presented here, large numbers of machines can easily be simulated. If necessary, temporary entities (jobs) can be entirely removed from the model (simply remove the single PUT and GET functions) without changing the behavior of the machines or the queue sizes. This would result in a vastly more efficient simulation which does not require any more memory when it becomes congested. Probably the best way to dramatically speed up a simulation of a manufacturing system is to use a resident entity event model like that given here. This model can be used to quickly focus on bottlenecks in very large systems. Transient entities can be introduced to measure job delay times anywhere in the system with PUT and GET functions.

ACKNOWLEDGEMENTS

Many SIGMA users have contributed suggestions that have been incorporated in the second release. Persons not acknowledged in [1] who have since contributed include Sheldon Jacobson, Douglas Morrice, Enver Yucesan, and Tony Zalewski.

Sheldon and Enver showed me an easy way to implement Perturbation Analysis in SIGMA, and Tony showed me a clever way to include optimization algorithms in SIGMA models. Although they were acknowledged in the SIGMA manual, Peter A. W. Lewis and Quint King have continued to be particularly helpful (Quint implemented SIGMA in Microsoft Windows). My earlier appreciation of David Briskman grows as I work with code he had written years ago when he introduced me to HOOPS graphics software [7]. The author's research on graph modeling and decomposition was supported by the National Research Council and the Naval Postgraduate School through their Senior Research Associates Program. Some of the results of that research will be included in the next release of SIGMA.

APPENDIX: SIGMA-GENERATED ENGLISH DESCRIPTION OF TANDEM QUEUE:

The SIGMA Model, TANDQ2.MOD, is a discrete event simulation. It models buffered tandem queues.

I. STATE VARIABLE DEFINITIONS

For this simulation, the following state variables are defined:

I:	index for loop reading input data (integer valued)
J:	index for stations (integer valued)
Q[10]:	number of jobs waiting at station j (integer valued)
B[10]:	waiting space for jobs at station j (integer valued)
S[10]:	jth server's status (free/busy/block=1/0/-1) (integer valued)
SYSTEM:	the global system queue (integer valued)
ENT[10]:	job attributes (ent[5]=time entered) (real valued)
RNK[6]:	ranking of the global system queue (fifo) (real valued)
MAKESPAN:	total job time in system (real valued)
N:	number of stations in sequence of queues (integer valued)
L[10]:	lower bound on uniform processing time at j (real valued)
U[10]:	upper bound on uniform processing time at j (real valued)

II. EVENT DEFINITIONS

Circumstances where the values of state variables might change are called events. State changes for this simulation are given below; these are represented by one or more event vertices (nodes or balls) in a SIGMA graph. Event parameters, if any, are given in parentheses.

The occurrence of an event often leads to further events. The logical and dynamic relationships between pairs of events are represented in a SIGMA graph by the edges (arrows) between event vertices. Unless otherwise stated, vertex execution priorities, to break time ties, are equal to 5.

1. The RUN(N) event models the initialization of the simulation. Initial values for, N, are needed for each run.
   After every occurrence of the RUN event:
   Unconditionally, read in the input data from tandq.dat; therefore, immediately execute the INPUT(I,J) event.using the parameter value(s) of 1,1.
   Unconditionally, initiate the first job arrival; therefore, schedule ARRIV() to occur without delay.

2. The ARRIV() event models the arrival of a new job.
   After every occurrence of the ARRIV event:
   Unconditionally, schedule the next arrival in about 10 minutes; therefore, schedule ARRIV() to occur in -10*ln{rnd} time units.
   If B[1]>Q[1],
   then there is room in queue 1 for the job; therefore, schedule ENTER() to occur without delay. (Time ties are broken by an execution priority of 3.)

3. The ENTER() event models the job entering the system (mark time). This event causes the following state change(s):
   ENT[5]=CLK
   After every occurrence of the ENTER event:
   If PUT{FIF;SYSTEM},
   then put the job in the first queue; therefore, immediately execute the WAIT(J) event...using the parameter value(s) of 1.

4. The WAIT(J) event models the job joining queue j. This event causes the following state change(s):
   Q[J]=Q[J]+1
   After every occurrence of the WAIT event:
   If S[J]>0,
   then the jth server is free, so start work; therefore, schedule START(J) to occur without delay...using the parameter value(s) of J. (Time ties are broken by an execution priority of 4.)

5. The START(J) event models the start of work at station j. This event causes the following state change(s):
   Q[J]=Q[J]-1
   S[J]=0
   After every occurrence of the START event:
   Unconditionally, work at station j; therefore, schedule DONE(J,S[J]) to occur in l[j]+u[j]*rnd time units...using the parameter value(s) of J,1.

6. The DONE(J,S[J]) event models the job finishing at station j.
   After every occurrence of the DONE event:
   If Q[J]>0 and (B[J+1]>Q[J+1] or J==N),
   then start working on the next job in queue j; therefore, schedule START(J) to occur without delay...using the parameter value(s) of J. (Time ties are broken by an execution priority of 2.)
   If J==N,
   then this is the last station, job is finished ; therefore, schedule FINSH() to occur without delay.
   If J!=N and B[J+1]>Q[J+1],
   then the job goes to the next queue ; therefore, schedule WAIT(J) to occur without delay...using the parameter value(s) of J+1. (Time ties are broken by an execution priority of 3.)
   If J!=N and B[J+1]<=Q[J+1] and S[J]!=-1,
   then server j becomes blocked, change status to -1; therefore, schedule DONE(J,S[J]) to occur without delay...using the parameter value(s) of J,-1. (Time ties are broken by an execution priority of 1.)
   If J!=1 and S[J-1]==-1 and S[J]!=-1,
   then free currently blocked server j-1; therefore, schedule DONE(J,S[J]) to occur without delay...using the parameter value(s) of J-1,1. (Time ties are broken by an execution priority of 6.)

7. The FINSH() event models the job finishing with the last station.
   After every occurrence of the FINSH event:
   If GET{FST;SYSTEM},
   then get the job from the system queue so it can leave; therefore, immediately execute the LEAVE(MAKESPAN) event...using the parameter value(s) of CLK-ENT[5].

8. The LEAVE(MAKESPAN) event models the job leaving the system. No additional events are scheduled here.

9. The INPUT(I,J) event models the data input (reads tandq.dat). This event causes the following state change(s):
   L[J]=DISK{TANDQ.DAT;I}
   U[J]=DISK{TANDQ.DAT;I+1}
   B[J]=DISK{TANDQ.DAT;I+2}
   S[J]=1
   After every occurrence of the INPUT event:
   If J<N,
   then read the input data for the next queue; therefore, schedule INPUT(I,J) to occur without delay...using the parameter value(s) of I+3,J+1. (Time ties are broken by an execution priority of 4.)

REFERENCES

[1] Schruben, L., and Yucesan, E. (1988), "Simulation Graphs," *Proc. 1988 Winter Simulation Conference.*

[2] Schruben, Lee, *SIGMA: A Graphical Simulation System*, (Release 2), Scientific Press, San Francisco, 1991.

[3] Schruben L. (1983), "Simulation Modeling with Event Graphs," *Comm. of the A.C.M.* 26(11).

[4] Hoover, Stewart, and Ronald Perry (1989) *Simulation: A Problem Solving Approach*, Addison-Wesley.

[5] Law, Averill and Kelton, W. David (1991), *Simulation Modeling and Analysis (2nd Ed.)*, McGraw Hill.

[6] Pegden, C. Dennis (1988), *Introduction to SIMAN, 2nd ed.*, Systems Modeling Corp.

[7] HOOPS—Hierarchical Object Oriented Picture System, Ithaca Software, San Francisco, CA.

AUTHOR BIOGRAPHY

LEE W. SCHRUBEN is on the faculty of the School of Operations Research and Industrial Engineering at Cornell University in Ithaca, NY. During the first half of 1991 he was a visiting professor at the Naval Postgraduate School in Monterey, California, supported by a National Research Council Senior Research Associateship.