Proceedings of the 1990 Winter Simulation Conference
Osman Balci, Randall P. Sadowski, Richard E. Nance (eds.)

SIMULATION GRAPHICAL MODELING AND ANALYSIS (SIGMA) TUTORIAL

Lee W. Schruben

School of Operations Research and Industrial Engineering
Cornell University
Ithaca, New York 14853-7501

ABSTRACT

SIGMA (abbreviated simply as Σ) is an interactive graphics approach to teaching discrete event simulation. Σ is specifically designed to make learning the fundamentals of simulation modeling easy. Σ can automatically translate a simulation model into Pascal or C source code that can be compiled and run on a wide variety of computers. It is possible to represent systems in all of the conventional discrete event world views with Σ. The viewpoint is the modeler's choice not a dictate of the language. Σ combines the modeling advantages of using network flow process and logic diagrams with the generality and flexibility of explicit event scheduling. The complete source code for Σ-generated simulation models is available to students. Although Σ is elementary, it is completely general; any discrete event simulation (indeed, any computer program) can be created using Σ.

1. INTRODUCTION

In Σ, stochastic discrete event systems are represented as simple dynamic graphs. Models are created by drawing these graphs with a mouse. Despite the fact that models are easy to create, Σ does not insulate students from the details of simulation model building.

Σ is written in C using the HOOPS object-oriented picture system from Ithaca Software of San Francisco, CA. However, Σ is completely self-contained and does not need a special compiler or graphics software.

If a compiled program is desired, with a single mouse click, a Σ model is automatically translated into portable standard C or Pascal (Turbo Pascal V5.5) 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 Σ-generated simulation is available. A student using Σ has access to, control over, and ultimate responsibility for all library functions for event list management, time advance, random variable generation, and output statistics collection. Since all of the Σ-generated source code for their models is available, students can quickly understand how their programs work; there are no mystery functions.

With Σ, complex models are transparent since the graph is the language. Students quickly learn to create and read simulation graphs. This visualization of the model makes it easy to detect logic errors.

2. EVENT GRAPH MODELING

Pictorially, Σ uses the vertices (nodes) of a graph to represent state changes that are associated with the various events in the simulation. The single building block of a Σ graph is a directed edge (arc) showing the relationships between pairs of events. Students do not need to learn dozens of specialized network symbols. To illustrate: suppose the edge below is part of the graph.

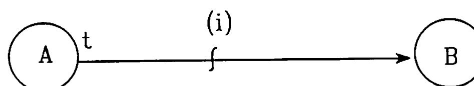

Figure 1. Conditional Event Graph Edge with a Delay Time

We interpret this edge:

> *"whenever event A occurs, if condition (i) is true, then event B will be scheduled to occur in t minutes."*

Basically, edges represent the conditions under which one event will cause another event to occur, perhaps after a time delay.

As an example of an event graph, we will consider a simple waiting line with a single server. The state variables we use to describe this system are

S = the status of the server
 = {idle,busy} = {1,0} and
Q = the number of customers waiting in line.

The event graph for this model is shown in Figure 2 where state changes are shown in parenthesis below each vertex.

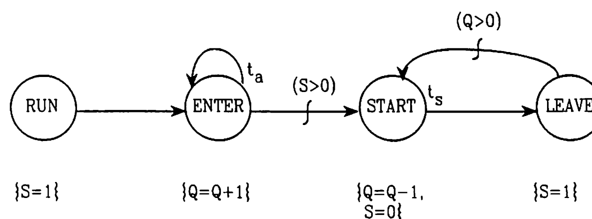

Figure 2. Single Server Queue with Edge Conditions and Delay Times

In addition to simply insuring that the events in a simulation model are tied together, one of the main advantages to simulation graph modeling is that each vertex can be isolated from the rest of the graph when describing it; the graph ties the events together. You "*read*" a simulation graph by successively describing the state changes at each vertex and the edges exiting that event.

The event graph in Figure 2 might be read as follows, with a single sentence describing each edge in the graph (try to identify the edge associated with each sentence).

The simulation RUN is started by making the server available (S=1) and having the first customer ENTER the system.

Each time a customer ENTERs the system, the queue length is incremented (Q=Q+1) and the next customer is scheduled to ENTER the system sometime in the future. If the ENTERing customer finds the server available, the server will START serving the customer right away.

When service STARTs, the server is made busy (S=0) and the length of the waiting line is decremented (Q=Q-1). The START service event will always schedule a customer to LEAVE when that customer completes service.

Whenever a customer LEAVEs the system, the server is made available (S=1) to serve the next customer. If there are still customers waiting in line, the LEAVE event will trigger the next service to START right away.

Now re-read the preceding four paragraphs without looking at Figure 2. You will see these paragraphs give a concise description of the behavior of a single-server queueing system. With practice, a verbal system description can easily be *read from the exiting edges* of a simulation event graph. This is an excellent way to communicate the essential features of a simulation model and a good first step in model validation. With experience in reading event graphs, it becomes relatively easy to detect modeling errors (and grade student's homework solutions!). While becoming familiar with reading event graphs, it might be a good idea to simply use the edge interpretation following Figure 1 as a template for describing each edge. State changes associated with each vertex in the graph are typically easy to check once the edges in the graph are correctly defined.

A Σ model of our queueing system is created by drawing a graph like the one in Figure 3 with a mouse. Figure 3 shows the completed model as it appears on the Σ screen. Details on the state variable changes associated with each event and the conditions and delays associated with edges are available by clicking the mouse on any object in the graph.

All the relationships between the events in this model were created with just 6 mouse clicks.

All commands in Σ are menu driven and there is extensive prompting. After reading a brief tutorial, students are able to start building and running discrete event simulation models right away. Although it takes getting used to, there are distinct advantages to developing a computer program on a plane as opposed to conventional linear coding. Students familiar with Σ have used it as a general purpose C or Pascal program prototyping tool irrespective of whether or not the program has anything to do with simulation.

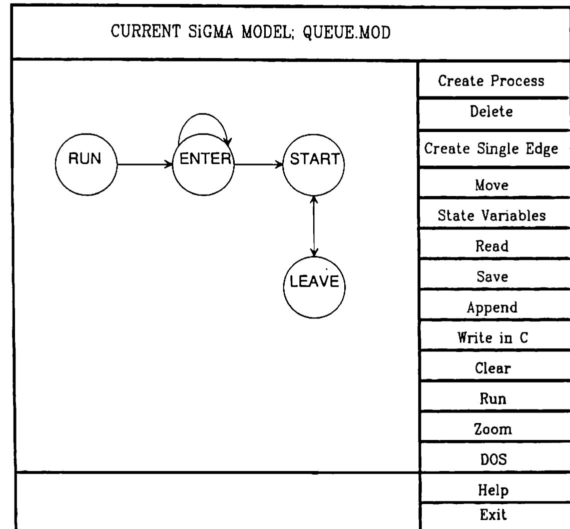

Figure 3. Event Graph for the Single Server Queue, QUEUE.MOD

3. SPECIFIC FEATURES OF Σ

Σ is designed to make learning simulation fundamentals fun. Several specific features of Σ are mentioned below.

Σ is Disposable: Since complete source code for a Σ-generated simulation model is available to the students, they can replace any of the functions with their own routines. Throughout the course, they can build their own fully functional "simulation languages."...no more mysteries!

Run Modes: Once the simulation graph is drawn on the screen it can be executed directly. Σ has three run modes: SINGLE-STEP run mode allows the student to step slowly through the simulation, watching changes in the state variables and future events list. Students graphically see the logic of their simulation models as it executes. GRAPHICS run mode uses color graphics to illustrate the dynamics of running the simulation model. Finally, there is the HIGH-SPEED run mode, which simply produces the output on a disk in a form suitable for spreadsheet analysis. With a spreadsheet program, it is a very easy matter to open a Σ output file and have it parsed and plotted in a variety of formats.

General USER functions: In addition to the usual arrays of integer and floating point variables, Σ has a general type of variable called a USER variable. A USER variable is actually an executable program. These functions can be written in any language that creates an executable file (eg: FORTRAN, C, Pascal, BASIC). As long as a simple calling convention is followed, these executable files do not even need to be linked with Σ. USER variables may be used in expressions like any other standard variable. This feature of Σ as well as the ease in debugging, makes Σ a useful tool for general program development, regardless of whether or not the program is a simulation.

Team Modeling: The Append command of Σ facilitates student team modeling. Students can individually model different components of a system which can then be "wired together" into a single simulation graph.

Parameterized Graphs: Σ has the simple but powerful modeling tool of event parameterization. An event parameter defines the particular system entities to which an event pertains. One application of event parameterization is the simultaneous running of parallel independent replications of the same simulation on the same CPU.

Automatic Source Code Generation: As mentioned earlier, with a single mouse click, a Σ model can be automatically translated into C or Pascal source code. Translating a Σ simulation graph into source code is very fast. Knowledge of C or Pascal is not necessary as Σ models can be run interpretatively without resorting to the generation of source code. However, Σ is a good introduction to programming, students learn about C or Pascal by seeing the generated source code, much the same as learning a foreign language by hearing it spoken. This is particularly true if a source level language environment is used, such as provided with Microsoft's QuickC or Codeview or Borland's Turbo C and Pascal.

Most commercial simulation languages isolate the student from the detailed operation of their models; in fact, source code is never available to the students. Furthermore, since these languages are very sophisticated, students spend much of the course learning language syntax and specialized tricks. In contrast, most popular textbooks on discrete event simulation focus on the fundamentals of simulation programs, which are not accessible to students using a commercial language. With Σ, students can learn and apply all of these modeling fundamentals.

Event Reduction: It is possible to execute neighboring vertices without first scheduling them on the future event's list. Thus several vertices can be condensed into a single event. This feature tests the student's knowledge of discrete event simulation dynamics and offers them the option of defining their "events" in any manner that makes sense. The conventional notion of an event can contain several vertices and their exiting edges.

4. CONTRASTING Σ WITH OTHER LANGUAGES

Σ is a teaching tool and is not intended as a substitute for commercial simulation languages. In fact, using Σ interactively to study system dynamics is most efficient using interpretive program execution. This is often too slow for large-scale simulation experiments. However, a practical simulation model source code generator is included in Σ that has been used commercially by former students. The generated source code is created with an emphasis on clarity rather than attempting to use efficient but obscure data structures and algorithms for simulation modeling. Still, the Σ-generated code (perhaps with minor modifications) can execute fast enough to be competitive with most commercial simulation languages and in some cases is faster.

Benchmarking of languages is in general a deceptive art. It does, however, give one a "feel" for the relative efficiency of a language without actually coding and running examples. To avoid any appearance of scientific validity, the following study consists of only one student running one model.

As part of one credit course last summer, a Master's student at Cornell implemented PC versions of SLAMSYSTEM (from Pritsker and Associates), SIMAN (from Systems Modeling Inc.), GPSS-H (from Wolverine Software), and Σ on a 286 based machine.

The familiar "barbershop" model in the GPSS-H manual was used. The graph for the Σ model for this problem appears in Figure 4.

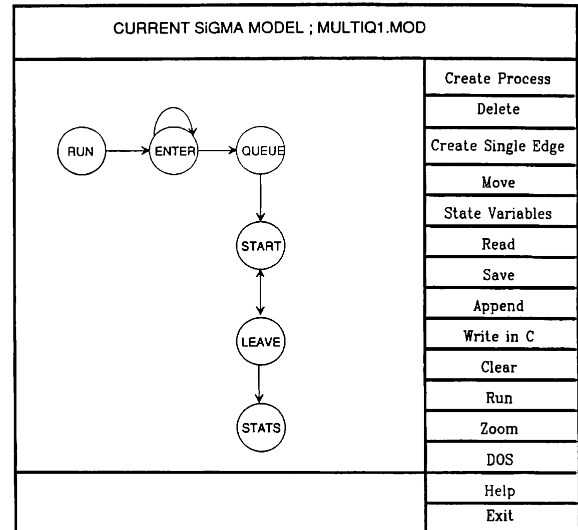

Figure 4. "Barbershop" Model

Network models for this or similar single-server queueing systems can be found in most language references. 1000 customers were run through the system, collecting waiting times for each customer. The following table gives some of the characteristics of the "barbershop" model implementations. All times were measured with an ordinary wristwatch.

	Network Symbols	Disk Space	Compile-Link Time	Run Time
SLAMSYSTEM	~25	5 Meg †	(Not Available)	27 Sec.
SIMAN/CINEMA	~50	6 Meg †	43 Sec.	32 Sec.
GPSS-H	~60	1 Meg	2 Sec.	27 Sec.
SIGMA	1			
Turbo Pascal		<.5 Meg	1 Sec	76 Sec.
Microsoft C (w/coprocessor)		<.5 Meg	19 Sec.	27 Sec. (6 Sec.)

† The full animation support for SLAMSYSTEM and SIMAN was implemented.

Table 1. A Benchmark Comparison for the Barbershop Problem

A process-oriented model was implemented in Σ; each customer was followed through the barbershop. A purely event-scheduling model of this system in Σ was considerably faster and more compact, but was also less informative.

The only remotely objective entry in the above table is in the first column. The fact that there is only one network symbol to learn with Σ means that students can begin detailed and general modeling in the first week of class. The instructor does not have to stick to the sequence of homework problems in a language text for fear that students do not yet know about a necessary block or network statement. After the first few lectures, the remainder of the semester can be spent covering simulation modeling and experimental fundamentals. Since complete source code for Σ-generated models is available to students, they should be able to figure out what is happening in their models.

As for the rest of table 1: the reader is again cautioned not to place too much trust in this (or any?) benchmarking study; merely note that the efficiency of Σ-generated code is competitive with that of other simulation tools... and it is much easier to learn and use.

ACKNOWLEDGEMENTS

SIGMA was developed completely without research funding. Persons who were interested enough in this project to volunteer their ideas, time, and talent include the following: David Briskman, Bob Covey, Daryl Goins, Kevin Healy, Tasha Henderson, Quint King, John McPeek, Christian Outzen, Charles Reily, and Brent Vallat. The people at Ithaca Software (of San Francisco, CA) have been helpful well beyond providing good support. Colleagues both at Cornell and elsewhere have stimulated this project with their contributions, encouragement, and challenges. These people include: David Heath, Bill Maxwell, Alan Pritsker, Steve Roberts, Paul Sanchez, Robert Sargent, Tapas Som, David Tate, and Enver Yucesan. Data for Table 1 was provided by Neville Shea, a former Master of Engineering and MBA student at Cornell.

AVAILABILITY: Σ is available, along with a illustrated user's guide, from Scientific Press, 651 Gateway Boulevard, San Francisco, CA 94080-7014. An instructor's disk with sample solutions to exercises in the user's guide and other examples is available free of charge to persons using Σ to teach courses at degree granting institutions.

SYSTEM REQUIREMENTS: The version of Σ described here requires an IBM PC compatible computer (AT or PS/2 preferred) with at least 640K of memory, a high-density floppy disk drive, a mouse, and an EGA or equivalent monitor with the corresponding graphics card. A hard disk is recommended. Other versions of Σ are currently being tested. Working prototypes have been developed for DEC MicroVax and Sun workstations.