Proceedings of the 1992 Winter Simulation Conference
ed. J. J. Swain, D. Goldsman, R. C. Crain, and J. R. Wilson

MODELING PRIORITY QUEUES WITH ENTITY LISTS: A SIGMA TUTORIAL

Lee W. Schruben

School of Operations Research and Industrial Engineering
Cornell University, Ithaca, New York 14853
and
SEMATECH, Austin, Texas 78741-6499 U.S.A.

ABSTRACT

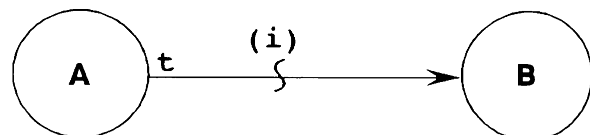
The use of ranked entity lists in SIGMA is illustrated in this paper. Previous Winter Simulation Conference tutorials illustrated the basic concepts of event graph modeling and the use of graph parameters to model very large systems. A complete introduction to the second release of the SIGMA graphical simulation modeling environment can be found in Schruben, 1992. After briefly introducing the single basic object for building event graph simulation models, an example is developed that illustrates the use of SIGMA entity lists. The example is of a time-constrained serial production process. Time-constrained sequences are very common in semiconductor manufacturing.

1. EVENT GRAPH MODELING

Event graphs can be used to build models of any discrete event system using just a *single* graphical object. This basic model building block consists of a directed edge (arrow) connecting a pair of vertices (balls). The vertices represent the changes in the values of system state variables that are associated with the occurrence of an elementary system event (such as the arrival of a customer to a queueing system). The edge between two event vertices represents the conditions under which one event might cause the occurrence of the other event as well as the time interval between the two events.

Associated with each event vertex is a set of state variable changes that take place whenever the corresponding event occurs. Event graphs are fundamentally different from the state transition diagrams commonly used to represent automata and Markov processes. The vertices in conventional state diagrams each represent a different *value* of the state; here vertices represent *changes* in state variables. Whereas the state diagram for a classical M/M/1 queue requires an infinite graph, the event graph for this system is not only finite but can actually be reduced to a single vertex!

Associated with each edge is a set of *conditions* that must be true in order for one event to schedule another. Also associated with each edge will be a *delay time* equal to the interval until the scheduled event occurs. The graphical representation of the basic modeling object is as follows;



This edge is interpreted as follows:

if condition (i) *is true at the instant event* A *occurs, then event* B *will be scheduled to occur* t *minutes later.*

If the condition is not true, nothing will happen, and the edge can be ignored until the next time event A occurs. You can think of an edge as nonexistent unless its edge condition is true. If the condition for an edge is always true (denoted as 1==1), the condition is left off the graph. We will call edges with conditions that are always true *unconditional* edges. Zero time delays for edges are not shown on the graph. By defining behaviors for different instances of the basic event graph object, different types of events can be modeled that can be linked into a complete system event graph model.

While learning to read event graphs, it is a good idea to use the edge interpretation above as a template for describing each edge. Once the edges in the graph are correct, the state changes associated with each vertex are typically easy to check.

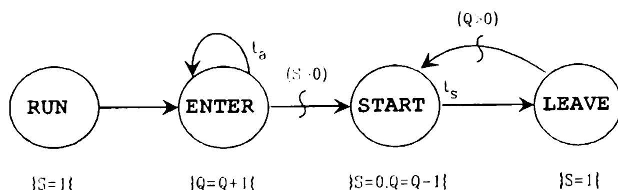
A key enrichment to event graphs is the concept of parameterized vertices and edge attributes that permit a basic event graph to be used to represent different

instances of similar subsystems. These graphs can be hierarchically linked into a model of a larger system. For example: an event graph of a single generic machine cell can be parameterized to represent different types of cells which can be linked in a larger graph that simulates a complete factory. Thus a generic factory simulator can be developed to any degree of complexity.

SIGMA is a point-and-click program for easily constructing event graph models. Other event graph enrichments included in the second release of SIGMA are interactive debugging, graphical output analysis, event canceling edges, automatic code generation in C, Pascal, or FORTRAN (SLAM and SIMAN) source code, and automatic generation of an English system description for model documentation and verification.

2. A SIMPLE EXAMPLE

A simple event graph is the model for a single-server queue. The state variable S will represent the status of the server (idle=1, busy=0) and Q the number of customers waiting in line. The times between successive arrivals (probably random) are denoted by t_a and the time required for service (perhaps also random) is denoted by t_s. When values of t_a or t_s are actually needed during a simulation run, they might be read from a data file or generated by random variable functions included in SIGMA. An event graph model for this system is as follows:



State changes associated with each elementary event are enclosed in braces below the respective vertices. Edge conditions appear in parentheses. A concise description of the system is obtained by simply describing each edge in the graph. In the following system description there is a 1 to 1 relationship between edges in the graph and sentences. (A useful exercise is to identify the edge in the event graph corresponding to each of the sentences in the following system description.)

At the start of the simulation RUN, the first customer will ENTER the system. Successive customers ENTER the system every t_a *minutes. If an ENTERing customer finds the server available* (S> 0), *they START service immediately. Customers who START service can LEAVE after a service delay of* t_s *minutes. Whenever a customer LEAVEs and the queue is not empty* (Q > 0), *the server will START with the next customer.*

This graph represents a completely defined simulation model. To run this model, only the starting and ending conditions for the run need to be specified.

If you now re-read the above paragraph without looking at the graph, you will see that it is a concise description of the behavior of the queueing system. With practice, a system description can be read easily from the 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 is becomes easier to detect logic errors in a model.

It is worth noting that while there is no universally accepted definition of an "event", the customary notion of a system event will typically correspond to a subgraph of event vertices connected by edges with zero delay. As mentioned earlier, if the arrival and service times for the above queueing system are exponentially distributed, the event graph can be collapsed into a single vertex representing the departure of a customer (i.e., representing the embedded Markov chain at customer departure times).

The ability to identify the events in a discrete event system is an important skill, one that takes practice to acquire. Initially, you might use the following simple steps as a guide to identify system events:

1. Identify the entities in your system.
2. Identify the dynamic attributes of each entity.
3. Identify the circumstances that *might* cause attribute values to change...these will be the events.

Once the elementary system events are identified, the process for constructing an event graph is rather straightforward. The rest of this paper will focus on the entity list tools included in the second release of SIGMA, followed by an example.

3. ENTITY LIST MANAGEMENT

Ranked queues occur whenever the order of service might differ from the order of customer arrival. SIGMA has two functions that make it very easy to model priority ranked queues and lists: The PUT function puts entries onto lists and the GET function gets entries off of lists. Two arrays, ENT[] and RNK[], are used by the PUT and GET functions. We will describe the purpose of these two arrays first.

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 temporary data buffer for customers joining ranked queues using the PUT{} and GET{} functions described below.

The array RNK[LINE] contains the index of the element of the ENT[] array that is to be used in determining a customer's position in the line designated by the integer, LINE.

The PUT{OPTION;LIST} function, places the current contents of the ENT[] array in the LIST. The elements of the temporary buffer, ENT, are typically the values of attributes of customers that are joining the queue. LIST is a number, variable, or function that identifies the queue to be joined. PUT{} OPTIONS include:

1 or FIF (first-is-first) inserts the new entity after the last record on the LIST.

2 or LIF (last-is-first) inserts the new entity before the first record on the LIST.

3 or INC (increasing) the LIST is ranked by increasing values of ENT[X], where X=RNK[LIST] is the ranking entity attribute.

4 or DEC (decreasing) the LIST is ranked by decreasing values of ENT[X], where X=RNK[LIST] is the ranking entity attribute.

5 or EVN (even) when the values of ENT[0] for two entities are even, the tie is broken by increasing values of ENT[2], with remaining ties broken FIFO.

The GET{OPTION;LIST} function removes a record from the specified LIST according to the OPTION chosen and places its contents in the ENT[] array. GET{} OPTIONs include:

1 or FST (first) removes the first entry of LIST.

2 or LST (last) removes the last entry of LIST.

3 or KEY (key) removes the first entry of LIST (if any) where values of ENT[0] match.

Both PUT{} and GET{} return 1 if successful and 0 otherwise. If these functions are used in a state change, they should appear only on the right-hand side of an equation, such as

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

Since the PUT function will return a value of 1, 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 be removed later (with attributes placed in ENT[]) with the state change,

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

Again the GET function will return a value of 1 if list N is not empty; thus, QUEUE[N] will be decreased by 1 by the above state change.

4. TIME-CONSTRAINED PROCESSING

The SIGMA ranked list functions, PUT and GET, are illustrated with an example of time-constrained processing. Consider a heat and press sequence where parts arrive every t_a minutes and must be heated in a furnace (which takes t_h minutes) before being pressed by a mold press machine (taking t_p minutes). The interarrival times and both processing steps are somewhat random. The sequence is time-constrained because there is a maximum cooling time limit of t_c after a part is heated during which the pressing step must start. If a part waits longer than t_c minutes after being heated before pressing starts, it must be returned to the furnace for reheating. We will identify each part waiting for the mold press with an ID number that is sequentially assigned after it is finished being heated. F and P will denote the status of the furnace and press respectively (1=idle, 0=busy). QF and QP will denote the number of parts in the queue waiting for the furnace and press. See Figure 1 for an event graph model of this system.

The graph in Figure 1 is the complete simulation model. It might seem rather complicated at first glance; however, one of the most appealing features of event graph modeling is that you do not have to look at the entire model at once. The logic in each vertex can be verified in isolation. The graph naturally decomposes into vertices and sets of exiting edges which can be examined separately; the graph ties everything together. We read the model in Figure 1 by examining one vertex at a time along with its set of exiting edges; ignore the rest of the graph as you read each of the following

paragraphs describing how each vertex in this model works.

The **RUN** vertex makes the furnace and press initially idle and requests a value for the cooling limit, t_c. The first part is then scheduled to **ENTER** the system.

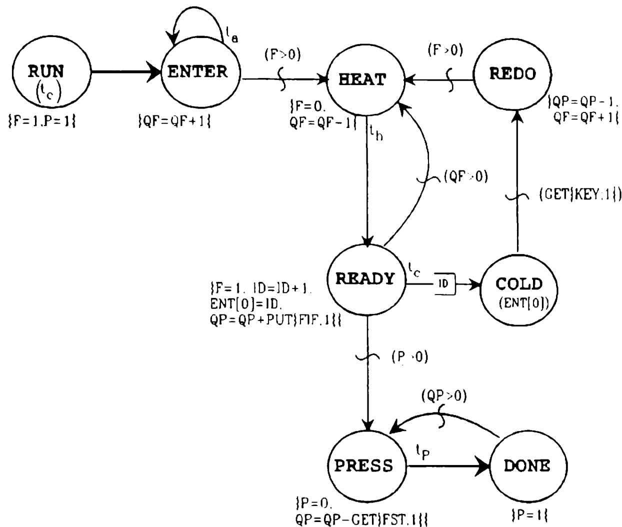


Figure 1: Event Graph of a Time-Constrained Sequence

When a new part **ENTERs** the queue for the furnace, QF is incremented. If the furnace is idle ($F>0$) the part can be **HEAT**ed. The **ENTER** vertex also schedules successive new part arrivals.

When **HEAT**ing starts, the furnace becomes busy ($F=0$) and the number of parts waiting for the furnace, QF, is decremented. After a heating time of t_h the part is **READY** for pressing.

When a part is **READY** for the mold press, the furnace is unloaded ($F=1$), an ID number is assigned to the part and it is PUT into the queue for the mold machine, QP. If there are more parts waiting for the furnace, the next part can start to be **HEAT**ed. If the mold press is idle ($P>0$), then the **PRESS** operation can begin immediately. Cooling starts as soon as the part is **READY** for the press. If part ID does not start its **PRESS** operation before t_c it will become **COLD** and need to be sent back to the furnace.

When the **COLD** event occurs, the cooling limit for part ID has been reached. The condition on the edge from **COLD** to **REDO** checks if the cold part is still waiting to be pressed using the GET function with the KEY option. If this part is not still waiting in the mold press queue (line 1), then it must have already started its pressing step. In this case, the GET function will not find a match and returns a value of 0 making the edge condition false and the REDO event will not be scheduled. Note that the ID *value* of the part is passed from the **READY** vertex to the **COLD** vertex as an edge attribute (see Schruben 1991,92). This ID *value* is placed in ENT[0] to be used as the match KEY in the GET function to see if that particular part is still waiting for the mold press when it became cold.

If the cold part is still in the queue for the mold press (that is, part ID is still waiting in list 1), then it is removed from the press queue ($QP=QP-1$) in the **REDO** vertex and placed back in the queue for the furnace ($QF=QF+1$). If the furnace is idle then **HEAT**ing can begin.

The **PRESS** operation will GET a part out of the queue (if the **COLD** event has not already removed it) and make the press busy. After a pressing time of t_p the part is **DONE**. If more parts are in the mold machine queue ($QP>0$) the next part waiting (that has not cooled off) can begin its **PRESS** step.

5. CONCLUDING REMARKS

In SIGMA, event graphs are drawn with a mouse. The model can then be interactively tested using powerful debugging aids such as the ASK function (Schruben, 1992). The model can also be interactively executed or translated into English, C, Pascal, or FORTRAN (SLAM or SIMAN). The C code generated by SIGMA for this simulation ran considerable faster on a PC under DOS than did models of this system developed in other languages. The full C source code is included so the simulation can easily be ported to other computers.

Event graphs can also be included in a larger model using the "append" command in SIGMA. In this way a large model can be developed by connecting its component sub graphs. This facilitates team model development. Even though all variables are global, only data is passed between subgraph objects in the form of event scheduling messages.

A completely new version of SIGMA for Windows is currently in beta test. The PC version of SIGMA running under DOS is available from Scientific Press, 651 Gateway Blvd., Suite 1100, South San Francisco, CA, 94080-7914 or by calling (415) 583-8840.

ACKNOWLEDGMENTS

I appreciate the opportunity to join SEMATECH during 1992 and have benefited from many enlightening discussions with the staff of the Modeling and Statistical Methods Group. In particular, the insights of Dr. John Fowler (who suggested the example in this paper) have been valuable. Most of all, I appreciate the students in the my simulation courses who have contributed to the development of SIGMA

over the past nine years and The Scientific Press for helping me sharpen the software and manual. The previous version of the SIGMA program was selected last year as one of the all-time best 101 educational computing applications by EDUCOM. Thank you.

REFERENCES

Schruben, Lee, 1990-91, Sigma Tutorials, *Proceedings of the Winter Simulation Conference.*

Schruben, Lee 1992, *Event Graph Modeling Using SIGMA*, (2nd release), The Scientific Press, S. San Francisco, CA.

AUTHOR BIOGRAPHY

Lee Schruben, a Professor in the School of Operations Research and Industrial Engineering at Cornell University, is currently a Visiting Distinguished Professor at SEMATECH, the semiconductor manufacturing and technology research consortium located in Austin Texas.