2001-04

Basic Event Graph Modeling

Buss, Arnold

Simulation News Europe

http://hdl.handle.net/10945/45519

TECHNICAL NOTES

Basic Event Graph Modeling

Arnold Buss
Operations Research Department, Naval Postgraduate School
Monterey, CA 93943-5000 U.S.A.

Introduction

This paper is a brief introduction to Event Graph methodology. Event Graphs are a way of graphically representing discrete-event simulation models. They have a minimalist design, with a single type of node and two types of edges with up to three options. Despite their simplicity, Event Graphs are extremely powerful. The Event Graph is the only graphical paradigm that directly models the event list logic. There are no conceptual limitations to the ability of Event Graphs to create a simulation model for any circumstance. Their simplicity, together with their extensibility, make them an ideal tool for rapid construction and prototyping of simulation models. Discrete Event Simulation

We assume the reader is familiar with the basic concepts of discrete event simulation (see any introductory text such as Law and Kelton 1991), so we will only briefly review the basics. Two fundamental components of a discrete event simulation model are a set of state variables, and a set of events. The model emulates the system being studied by producing state trajectories, that is, the time history of successive values of the system's state variables.

Measures of performance are computed as statistics based on these state trajectories. Discrete event models are characterized by state trajectories that are piecewise constant. Events occur at the points in time at which at least one state variable changes value. It is important to note that an event is an instantaneous occurrence in the discrete event model. No simulated time passes when an event occurs; simulated time passes only between the occurrence of events.

The timing of the occurrence of events is controlled by the Future Event List (or simply the Event List), which is nothing more than a “to-do” list of scheduled events. Whenever an event is scheduled to occur, an event notice is created and stored on the future events list. Every event notice contains two pieces of information: (1) what event is being scheduled; and (2) the (simulated) time at which the event is to occur. The Event List keeps the event notices in order by ranking them based on the lowest scheduled time. Events occurring simultaneously in simulated time must be prioritized according to some secondary rule. The future events list is managed by basic discrete event algorithm that controls the flow of time in the simulated world of the model.

At each iteration the algorithm examines the event list to see if there are any scheduled events. An empty list means there is nothing to do, so the simulation terminates (i.e. the simulation run ends). If the event list is not empty, the simulated clock is updated to the time of the first event and the associated event is executed - that is, the state transitions associated with that event are performed. Note that the terminating condition (empty event list) means the simulation must be initiated with at least one scheduled event for any event to actually occur. We follow Schruben (1995) by identifying one distinguished event (Run) that is always on the event list initially. The Run event is responsible for scheduling the initial events of the model.

By convention, when an event occurs, all state changes associated with that event are first performed. Next, all further events are scheduled, and finally the event notice is removed from the Event List. The events scheduled are specified by the occurring event itself and may be conditional on certain values of the current state. The order of execution for these three steps could be altered and different, but equivalent, models would result. Although it is possible to perform the actions in arbitrary order (e.g. First change some states, then schedule some events, then change some more states, etc.), the resulting models would be confusing and error-prone. There is considerable benefit from adapting a convention such as the one above.

Event Graphs

Event Graphs are a way of representing the Future Event List logic for a discrete-event model. An Event Graph consists of nodes and directed edges. Each node corresponds to an event, or state transition, and each edge corresponds to the scheduling of other events. Each edge can optionally have an associated boolean condition and/or a time delay. Figure 1 shows the fundamental construct for Event Graphs and is

interpreted as follows: the occurrence of Event A causes Event B to be scheduled after a time delay of t, providing condition (i) is true (after the state transitions for Event A have been performed). By convention, the time delay t is indicated toward the tail of the scheduling edge and the edge condition is shown just above the wavy line through the middle of the edge. If there is no time delay, then t is omitted. Similarly, if Event B is always scheduled following the occurrence of Event B, then the edge condition is omitted, and the edge is called an unconditional edge.Thus, the basic Event Graph paradigm contains only two elements: the event node and the scheduling edge with two options on the edges (time delay and edge condition).

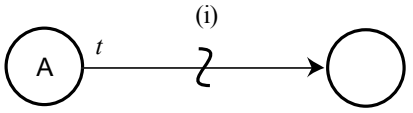


Figure 1. Fundamental Event Graph Construct

The simplicity of the Event Graph paradigm is evident from the fact that we can represent any discrete event model using only these constructs (Schruben 1992, 1995; Schruben and Yücesan 1993). An advantage of the minimalist approach of Event Graphs is that the modeler can spend more time on model formulation and less on learning the constructs of the paradigm.

There is a price to the simplicity of Event Graphs, however. Since Event Graphs represent the event scheduling relationship, rather than the physical movement of, say, customers through a queueing system, Event Graphs require a higher degree of abstraction on the part of the simulation modeler than the more commonly used process/resource world view. The author's experience using Event Graphs in an introductory simulation course is that the higher abstraction of Event Graphs is easy to master and provides rich payoffs for understanding and creating discrete event simulations. Indeed, the use of Event Graphs tends to accelerate the understanding of the Discrete Event paradigm.

Example

The simplest non-trivial Event Graph is the Arrival Process, a model with a single event (Arrival) and a single state variable, the cumulative number of arrivals (N). The time between arrivals is modeled as a sequence of interarrival times $\{t_A\}$ that can be constant, a sequence of iid random variables (making the model that of a renewal process), or any arbitrary process of non-zero numbers. The state transition for the Arrival event is that the cumulative number of arrivals (N) be incremented by 1. The Event Graph for the Arrival Process is show in Figure 2.

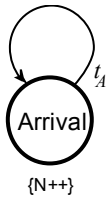


Figure 2. The Arrival Process Event Graph

Since the Event List is initially empty, the terminating condition for the simulation run, there must be at least one event scheduled initially. Event Graphs provide this by means of a bootstrapping event called "Run." The Run event is placed on the Event List at time 0.0 but is otherwise an ordinary event with associated state transitions and scheduling edges. Thus, to make the Arrival Process model in Figure 2 a complete running model, a Run event is added that simply initializes the cumulative number of arrivals to 0 and scheduled the first arrival, as shown in Figure 3.

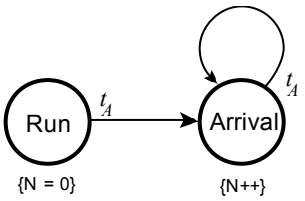


Figure 3. Arrival Process with Run Event

Simultaneous Events

There is one difficulty in the straightforward application of the Event Graph methodology presented so far to complex models, namely that of resolving the execution order of simultaneous events. Simultaneous events occur when more than one event is schedule to occur the exactly the same time. In some cases the order of execution of the events is irrelevant, but in other cases certain permutations of the order of occurrence impact the outcome dramatically, often leading to invalid state trajectories and inadmissible values of state

variables. Since computers have finite precision, this possibility cannot be discounted even when "continuous" random variables are being used. For the simple model in Figure 3 there is no problem with simultaneous events, but even in slightly more involved models (such as the queueing model discussed in the following section) the problem of resolving simultaneous events arises. If discrete probability distributions are used to model delay times then the potential for simultaneous events increases dramatically.

Event Graph methodology provides the capability of prioritizing scheduling edges, so that simultaneous occurrences of the scheduled event always occur before other scheduled events. Although these edge priorities are typically not indicated on the graph itself, all software implementations of Event Graph methodology support edge prioritization.

Further Examples

We will now present some additional examples of Event Graph models. The first is a standard multiple-server queue.

Multiple Server Queue

Description.

Customers arrive to a service facility according to an arrival process and are served by one of k servers. Customers arriving to find all servers busy wait in a single queue and are served in order of their arrival.

Parameters

$\{t_A\}$ = interarrival times; $\{t_S\}$ = service times; k = total number of servers.

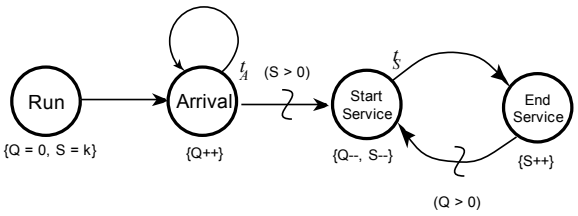
State Variables

Q = # of customers in queue; S = # of available servers

Event Graph

Comments

In this model, entity-oriented data (such as time in queue or time in system) are not explicitly available. The idea is that time-varying statistics can be collected on each state variable and these are ordinarily sufficient to compute any desired performance measure. In this case, Little's formula can be applied to translate time averages into tallied statistics for delay in queue or time in system.



This model bears a superficial resemblance to a process-oriented model, since the events correspond to the sequence of actions that occur as a customer proceeds through the system. However, close inspection shows that the scheduling edge going "backwards" from the EndService event to the StartService event do not have a direct correspondence in a process model. The Event Graph captures the scheduling dependencies of the events in the model, not the flow of customers or entities through the system. That is, the Event Graph does not represent a synchronous flow of event execution, but scheduling relationship between the various events which are executed asynchronously when the simulation is run.

For more flexible models, it is highly desirable to separate the arrival process from the server part of the model into two distinct components. The two components can be loosely-coupled to work together (see Buss, 2000). For this introductory note, however, we will confine ourselves to simple models with no component approach.

Tandem Queue

Description

Arriving customers are processed by one workstation consisting of a multiple-server queue. Upon completion of service at the first workstation, a customer proceeds with probability to a second workstation or departs the system with probability 1- p.

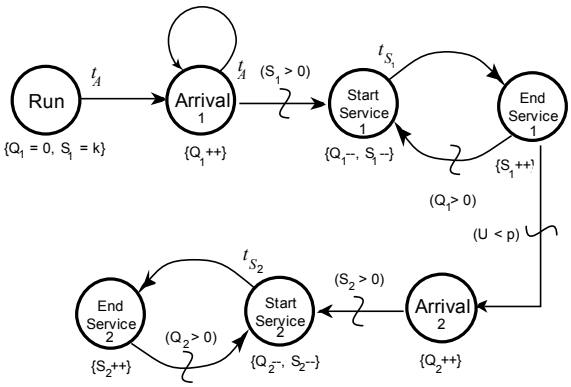
Parameters

$\{t_A\}$ = interarrival times; $\{t_{Si}\}$ = service times at workstation i (i=1,2); k_i = total number of servers at workstation i; p = probability of customer proceeding to second workstation; {U} a sequence of iid Un(0,1) random variables.

State Variables

Q_i = # of customers in queue for workstation i; S_i = # of available servers at workstation i.

Event Graph:



Comments

This model can easily be extended to models with any number of workcenters by appending more copies of the "Server" portion of the Event Graph. However, as the number of workcenters becomes very large, the resulting model becomes unwieldy. Process-oriented methodologies have the same difficulty scaling up. More scalable Event Graph models can be created in two ways: exploiting parameters on edges and events, discussed below, and the use of a component framework for creating "building blocks" consisting of relatively small Event Graph pieces Buss (2000).

Extensions

In principle the simple construct in Figure 1 is all that is needed to create any discrete event simulation models. In practice, however, there are two simple extensions that enhance event graph models' ease of use and enable much simpler models to be created. These extensions are the cancelling edge and the ability to pass parameters on edges.

Cancelling Edges

The cancelling edge is the inverse operation of the scheduling edge, and is represented in Figure 4.

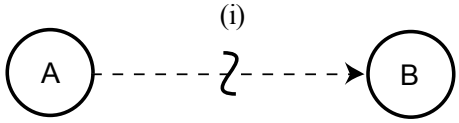


Figure 4. Cancelling Edge

The interpretation of Figure 4 is as follows. Whenever event A occurs, then if condition (i) is true, the first occurrence of event B is removed from the event list. If event B is not scheduled to occur, then nothing happens. If there are multiple occurrences, only the first is removed. The priority of the events is used to break ties when multiple events of the same type are scheduled to occur at the same time.

Example: Server With Failures

Description.

A machine is subject to periodic failures, which occur after a certain amount of time regardless of how long it has actually been in service. Upon failure, the part being processed (if any) is returned to the queue until the machine is repaired.

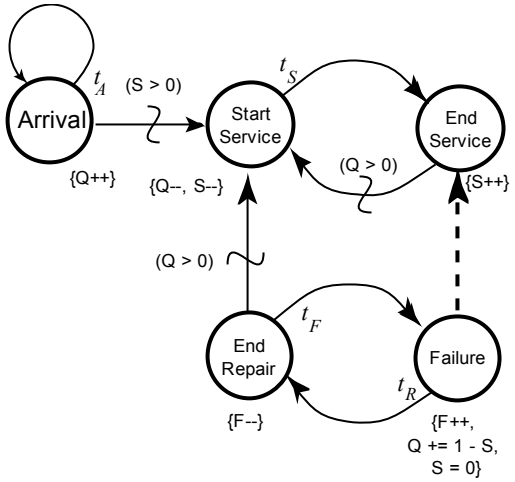
Parameters

$\{t_A\}$=the sequence of interarrival times of parts; $\{t_S\}$=the sequence of service times; $\{t_F\}$=machine times-to-failure; $\{t_R\}$=repair times.

State Variables

Q=# of parts in queue; S=1/0 if machine is available/busy; F=0/1 if machine is working/failed.

Event Graph.



Notes

The initialization of the Event Graph above has been omitted for clarity. Initially, there should be an Arrival event and a Failure event on the event list. Note how the condition for an Arrival event triggering a StartService event now is that the machine be both available and working. The priority order for simultaneous events is Failure < StartService < Arrival. That is, a Failure event will be performed before all other StartService and Arri-

val events that are scheduled at exactly the same time. Similarly, every StartService event will be performed before every other simultaneously scheduled Arrival event. Note that these priorities apply only when the events are scheduled to occur at exactly the same time. The other events do not need to be ordered.

Parameters on Edges

The second important extension is the ability to pass parameters on edges to event nodes. This is represented in Figure 5 for both scheduling and for cancelling edges.

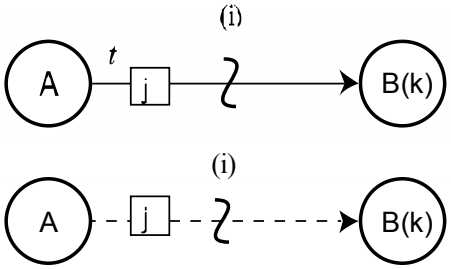


Figure 5. Scheduling and Cancelling Edges with Parameters

The interpretation of the constructs in Figure 5 as follows. For the scheduling edge with parameter: When Event A occurs then, if condition (i) is true, event B is scheduled to occur after a delay of t time units; when B occurs, its parameter k will be set to the value given by the expression j. For the cancelling edge with parameter: When event A occurs then, if condition (i) is true, the first scheduled event of type B whose parameter k exactly matches j is removed; if no such event is found, then nothing happens. When event B occurs, the value of expression j is that which it had when the scheduling event A occurred.

The relationship between the parameter on the event node and the matching parameter on the scheduling edge is the same as that between the code in a program that invokes a procedure with an argument and the argument of the procedure that matches the call. Thus, in Figure 5, j is an expression that resolves to a value only when event A occurs, whereas k is a formal parameter. The parameter can be considered a "time capsule," that is, a means of passing information about the current state of the model to a future event.

Example: Transfer Line

The capability of passing parameters on edges enables a generic model of multiple server queues in series to be created. To model a series of three or more workstations in a line by extending the tandem queue above would require modification of the Event Graph itself.. Instead, using parameters on edges, one event graph model can be developed that can model any number of workstations in a series based only on input data.

Description

Arriving customers are processed by n workstations in a series, each consisting of a multiple-server queue. Upon completion of service at each workstation, a customer proceeds to the next workstations and departs the system when service at the last workstation is complete.

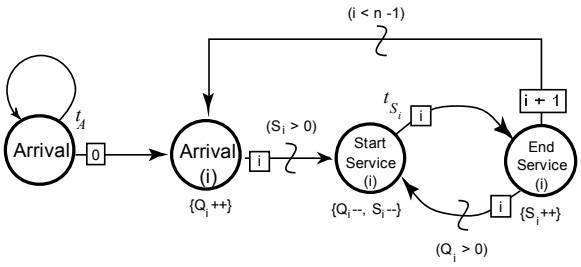
Parameters.

n = number of workcenters (numbered 0,...,n-1). $\{t_A\}$ = interarrival times of customers to the system; $\{t_{Si}\}$ = service times at workstation i; k_i = total number of servers at workstation i.

State Variables

Q_i = # of customers in queue for workstation i; S_i = # of available servers at workstation i.

Event Graph:



Comments

In this model, parts come into the system with the ArrivalToSystem event, which is distinct from the Arrival event which signifies the arrival of a part to a workcenter. The parameter on each scheduling edge is the workcenter for which the scheduled event is to occur.

Implementations

To the author's knowledge there are only two software packages that directly support building Event Graph models, Sigma™ and Simkit.

Sigma

Sigma™ is a windows program that allows the modeler to draw the Event Graph in a palette, then add state variables and parameters to events and edges in dialog boxes. The model can be executed graphically, and the standard set of statistics and plots generated. A very useful feature is the ability to generate the C code for the model so it can be run as an independent program.

Simkit

Simkit is a set of Java™ packages that support building discrete-event models from an Event Graph perspective. It does not currently have any built-in graphic capabilities, such as the Event Graph palette in Sigma. However, it is Open Source and freely available under the GNU Public License.

Simkit extends the basic Event Graph paradigm by adding a component architecture based on loose coupling of simulation components. More information on this approach can be found in Buss (2000). Simkit can be downloaded from the internet at the following URL:

http://diana.or.nps.navy.mil/simkit/

Conclusions

Event Graphs are a simple, yet powerful way to create Discrete Event Simulation models. Their simplicity makes them an excellent platform for teaching discrete event simulation, and their power makes them a good platform for building many different types of simulation model. The examples shown should give an indication of their usefulness in creating discrete event simulation models. Some simple extensions extend the flexibility and expressiveness of event Graph models. There are two software packages to assist the creation of simulation programs based on Event Graph models.

References

[1] Buss, A. 1996. Modeling with Event Graphs, Proceedings of the 1996 Winter Simulation Conference*, J. M. Games, D. J. Morrice, D. T. Brunner, and J. J. Swain, eds.*

[2] Buss, A. 2000. Component-Based Simulation Modeling, Proceedings of the 2000 Winter Simulation Conference, J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, eds.

[3] Law, A. and D. Kelton. 2000. Simulation Modeling and Analysis, Third Edition, McGraw-Hill, Boston. MA.

[4] Schruben, L. 1983. Simulation Modeling with Event Graphs, Communications of the ACM, 26, 957-963.

[5] Schruben, L. 1995. Graphical Simulation Modeling and Analysis Using Sigma for Windows, Boyd and Fraser Publishing Company, Danvers, MA.

[6] Schruben, L and E. Yücesan. 1993. Modeling Paradigms for Discrete Event Simulation, Operations Research Letters, 13, 265–275.

[7] Schruben, L. and E. Yücesan. 1994. Transforming Petri Nets Into Event Graph Models. Proceedings of the 1994 Winter Simulation Conference, J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila, eds.