2006

As simple as possible, but no simpler: A gentle introduction to simulation modeling

Sanchez, Paul J.

Sanchez, P. J. 2006. "As simple as possible, but no simpler: A gentle introduction to simulation modeling," Proceedings of the 2006 Winter Simulation Conference, 1-10.

Proceedings of the 2006 Winter Simulation Conference
L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, eds.

AS SIMPLE AS POSSIBLE, BUT NO SIMPLER: A GENTLE INTRODUCTION TO SIMULATION MODELING

Paul J. Sánchez

Operations Research Department
Naval Postgraduate School
Monterey, CA 93943, U.S.A.

ABSTRACT

We start with basic terminology and concepts of modeling, and decompose the art of modeling as a process. This overview of the process helps clarify when we should or should not use simulation models. We discuss some common missteps made by many inexperienced modelers, and propose a concrete approach for avoiding those mistakes. After a quick review of event graphs, which are a very straightforward notation for discrete event systems, we illustrate how an event graph can be translated quite directly to a computer program with the aid of a surprisingly simple library. The resulting programs are easy to implement and computationally are extremely efficient. The first half of the paper focuses principles of modeling, and should be of general interest. The second half will be of interest to students, teachers, and readers who wish to know how simulation models work and how to implement them from the ground up.

1 BACKGROUND & TERMINOLOGY

We use models in an attempt to gain understanding and insights about some aspect of the real world. Attempts to model reality assume *a priori* the existence of some type of "ground truth," which impartial and omniscient observers would agree upon. Let's start off by considering the universe—the universe, for our purposes, is the set of everything in existence over all time. While there are many fascinating possibilities for discussion about the totality of the universe, we set our sights much lower in this paper, observing that there is a much greater chance of finding consensus when we focus our attention locally in time and space. We will start our study of models at the level of a system. There are many excellent resources available for those who wish to study the topic of modeling in greater depth. See, for example, Law and Kelton (2000), Banks et al. (2005), Weinberg (2001), or Nise (2004).

We define a *system* to be a set of elements which interact or interrelate in some fashion. Elements which have no relationship to other elements which we classify as members of the system cannot affect the system's elements, and thus are irrelevant to our goal of studying the system. The elements that make up the system are often referred to as *entities*. Note that the entities which comprise a system need not be tangible. For instance, we can talk about a queueing system, which is made up of customers, a queue, and a server. The customers and server are physical entities, but the queue itself is a concept. In some cultures, people waiting for a bus mimic the concept by standing in a row. However, there are some cultures where no line forms but it is considered improper to board the bus until everybody who was there before you has done so.

Systems can exhibit set ownership or membership with regard to other systems. In other words, a given system can be made up of sub-systems, and/or may in turn be a sub-system within a larger framework.

A *model* is a system which we use as a surrogate for another system. There can be many reasons for using a model. For instance, models can enable us to study how a prospective system will work before the real system has even been built. In many cases, the cost of building and studying a model is a small fraction of the cost of experimenting with the real system. Models can also be used to mitigate risk—it is far safer to teach a pilot how to cope with wind sheer during landing on a flight simulator than by going out and practicing real landings in wind sheer conditions. Another benefit is a model's ability to scale time or space in a favorable manner—with a flight simulator we can create wind sheer conditions on demand, rather than flying around "hoping" to encounter them.

Models come in many varieties. These can include but are not limited to physical duplicates (with or without scaling) such as wind tunnel mockups; "clockwork" and cam devices such as the Antikythera mechanism (de Solla Price 1959) or fire control computers on pre-digital battleships; mathematical equations such as the equations of motion

found in a typical physics text; analog circuitry such as that found in old stationary flight simulators; or computer programs such as the ones used in modern flight simulators. A *computer simulation* is a model which happens to be a computer program. Throughout the remainder of this paper we will use the word "simulation" to mean computer simulation, but you should be aware that this may be a source of miscommunication when dealing with people from other disciplines.

In all cases, models have a common purpose—to mimic or describe the behavior of the system being modeled. In most cases models simplify or abstract the real system to reduce cost and/or focus on essential characteristics. In fact, most of the examples in the previous paragraph work by producing a system which mimics the behavior in an input/output sense, but not the actual workings of the system being studied. We should judge a model's quality by how well its outputs conform to observations of reality, rather than by the amount of detail included in the model.

In practice we like models which are comprised of model entities similar to those in the real system, and which interact and change in ways which correspond to the interactions and changes observed or expected in the real system. The totality of all entities and all of their attributes is the *state* of the system, so we seek to model the real system by specifying when and how the model state should change so as to correspond to state changes in the real system. If the real system is deterministic (i.e., has no random elements), we try to produce state trajectories which are similar to those of the real system. If the real system is stochastic, we do not need to match state trajectories directly. Instead, we try to produce state trajectories which are plausible realizations of what might be seen in the real system.

A model should be created to address a specific set of questions. Some people believe that it is possible to build a completely general model, which could later be used to answer any question. At first glance this is appealing, but after a little bit of thought it should be obvious that the only way to achieve this would be to have the model state space be as large as the real system's state. Only a replica of the original system, complete in every detail, would have the ability to answer any and every unanticipated question about the system. This is the very antithesis of modeling, since the purpose of modeling is to simplify and abstract to gain insights.

2 AN OVERVIEW OF THE MODELING PROCESS

In practice, modeling is an iterative process with feedback. We start by considering the real-world situation we wish to know more about. In stage 1 of the modeling process we should try to identify what is meant by the *system of interest*. For instance, suppose we want to model the operations of a manufacturing plant which makes small boats. In reality there may be airplanes or Canadian geese which fly overhead, but unless we're concerned about the impact of plane crashes or organic pollution we should not consider these to be elements of the system. Similarly, while raw materials, customer purchase orders, weather, and marketing strategies will undoubtedly have an impact on our system, if we are trying to figure out a good shop-floor layout these can be represented as exogenous inputs, i.e., inputs which are determined by forces outside the system. For example, we need a stream of weather data which is similar to what we might observe in reality, but we don't need a physics-based weather module which mimics atmospheric heat transfer, humidity, convection, solar reflectivity, etc. An historical trace of past weather patterns or a random variate generator which adequately mimics the distribution of observed weather will more than likely suffice. At the end of the stage 1 process, we have a *descriptive model*.

Once we have decided on the scope of our model, we will proceed to the next phase. In stage 2, we try to rigorously describe the behaviors and interactions of all of the entities which comprise the system. This can be accomplished in a variety of ways, many of which are mathematical in nature. We might describe the system as a set of differential equations, or as a set of constraints and objectives in some optimization formulation, or use distribution modeling from probability or stochastic processes. We refer to the result as a *formal model*.

We would like to find analytical solutions to the formal model if it is possible to do so. If our formal model has a high degree of conformance with the real world system being modeled, analytic models and their solutions would allow us to obtain insights and draw inferences about the real system (see Figure 1). It is all too often the case, however, that in our quest for a good model we add components which make the formal model intractible. For example, we can and should find analytic solutions for queueing systems where arrival and service distributions are exponential with constant rates. Adding real-world features in the form of other distributions, non-homogeneous (possibly state-dependent) arrival and service rates, customers jockeying or balking, servers taking breaks, machinery breaking down, and so on, will very quickly put us into the realm of models which cannot be solved analytically.

This is one of the places where simulation might enter the process. In many cases we can describe the behaviors in a system algorithmically, producing a computer simulation as our model. If the simulation model uses randomness as part of the modeling process, its output is a random variable. A very common (and extremely serious!) mistake that first-time simulators make is to run a stochastic model one time and believe that they have found "the answer." The proper way to describe or analyze a stochastic system is with statistics. In other words, we must build a statistical

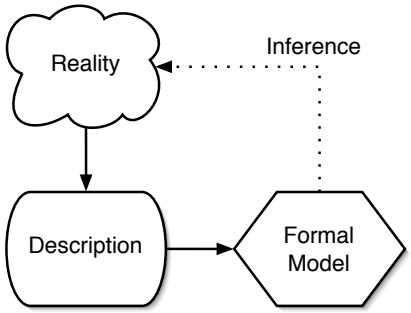

Figure 1: A Model Yields Insights & Inferences

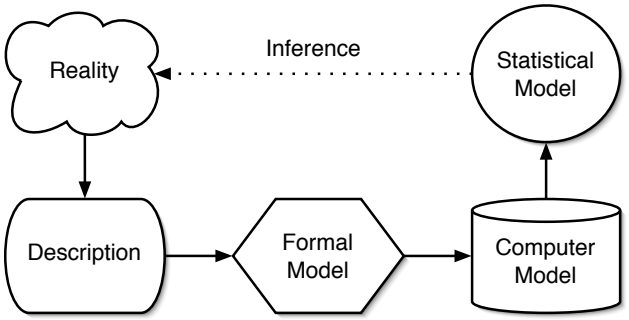

Figure 2: Simulation Has a Longer Chain of Inference

model of the computer model we built from the formal model. The resulting process is illustrated in Figure 2.

Feedback enters the modeling process in the form of *verification* and *validation* (Sargent 2003). Verification constitutes a feedback loop between the computer model and the formal model in Figure 2. In essence it attempts to address the question "does my computer program do what I meant it to do?" The formal model is the expression of your intent. Verification corresponds to the computer science task of debugging, which is considered a very hard problem indeed. However, validation poses an even more challenging question—"does my computer program mimic reality adequately?" Validation constitutes a feedback loop between the computer model and reality. It should be clear that the verification feedback loop is contained within the validation loop, i.e., you cannot talk about validating a model until you believe that it properly reflects your intent. In general you can expect to go through multiple iterations of verification and validation before you are satisfied with your model.

In both Figures 1 and 2 the solid arrows represent phases in the modeling process in which we move from one stage to another, with all of the associated simplifications, assumptions, and distortions that are introduced by the very act of modeling. Comparing the two figures, the process of doing simulation involves more stages, and therefore more opportunities to mess things up. Simulation modeling involves a longer chain of inference than does analytical modeling, which is why we generally would prefer to use analytical solutions where possible.

As with most rules, there are exceptions. For example, simulation can be an ideal technology for validating new processes or procedures. Suppose that you wish to demonstrate the superiority of a new statistical technique which you claim is optimal when the data follow a particular distribution. With observational data you can do goodness of fit tests to check for the desired distribution, but in practice such tests have notoriously poor discriminating power. With simulation you can guarantee the distribution of the inputs.

We'll finish this section with the following recommendations to modelers. Many modelers make the mistake of equating detail with accuracy. They start with a grand vision of a highly detailed model which mirrors every aspect of the real world system. As a result they may run out of time or budget before they ever get their model running. Those who manage to create a running program end up with code bases often measured in tens-, hundreds-, or even higher multiples of thousands of lines of code. The sheer magnitude of such programs makes verification and validation nearly impossible. The behavior of the program is determined by dozens to hundreds of IRKs (Independent Rheostat Knobs), inputs whose correspondence to reality is tenuous at best and which unethical analysts have been known to use to "tune" a model to produce desired outcomes.

It will not surprise the astute reader to note that we advocate a different approach.

- Start small - Begin with the simplest possible model which captures the essence of the system you wish to study.
- Improve incrementally - Once you have a basic model working, you can add features to it to improve the representation of reality. However, do so in small steps. Try to prioritize your additions in terms of greatest anticipated improvement in the model.
- Test frequently - The objective is a model which conforms well to reality, not one which is a duplicate. After each of your incremental improvements, check the resulting model. Does it do a better job of modeling? Did the new addition break anything?
- Do not be afraid to backtrack or simplify - There comes a point where you face diminishing returns. Sometimes, an addition produces no measurable benefit. Do not be afraid to chuck it out if it adds nothing but complexity.

Using this approach you are more likely to achieve a functioning model. If you are constrained on budget or time, you will still have built the best model which could be achieved within these constraints. If you have reached the point of diminishing returns on model investment, you produce

a model which produces answers as good as (and possibly better than) those of more complex models, without the complexity. Either way you will have built the most economical model for your purposes.

3 DISCRETE EVENT SYSTEMS

There are many classifications of systems available. The Winter Simulation Conference tends to focus on *Discrete Event Systems*. These are systems where the state changes occur at a discrete set of points along the time axis, rather than continuously. The points in time corresponding to state changes are called *event*s. *Discrete Event Simulation* (DES) models can be built with any of several world views (Nance 1981).

Much of the simulation software which is commercially available uses the *Process* world view for modeling. Process models are considered to be very accessible—the modeler describes the sequence of resource requirements, activities, delays, and decisions that an entity experiences as it proceeds through the system from start to finish. The details of how this is accomplished are similar but specific to each simulation package.

Event scheduling is another world view which can be used to construct DES models, and yields efficient implementations quite straightforwardly when the model is to be written in a lower level language. DES works by advancing simulated time directly from one event to another. Intervals of time between events are of no interest, because by definition nothing is happening during those intervals. Schruben (1983) created *event graph* notation so that simulation modelers could focus on the model-specific logic of the system to be studied. Event graphs provide a concise, unambiguous description of both how events change the system state and how they trigger the occurrence of further events.

Let's talk briefly about another type of error that modelers can make. An old joke that says "to the man who only owns a hammer, all problems look like a nail." The modeling equivalent of that joke is no joke at all. It is a concept called a *Type III error* by Mitroff and Featheringham (1974), who defined it as "the error…[of] choosing the wrong problem representation…" This can happen, as in the joke, when the analyst tries to fit the problem to the tool rather than vice-versa. You are at risk of committing a Type III error when you find yourself trying to "trick" your software into performing some modeling task.

Simulation languages are an example of what computer scientists call *Domain Specific Languages* (DSLs) (Mernik, Heering, and Sloane 2005). DSLs are very good at expressing problems within their chosen modeling framework and domain of expertise, but become intractable outside those boundaries. The alternative to DSLs is *General-Purpose Programming Languages* (GPPLs). By definition GPPLs are *Turing complete* (Brainerd and Landweber 1974), which means that if a problem is computationally feasible it can be expressed in a GPPL.

Many people shy away from using GPPLs for simulation, either because they do not know how simulation actually works or because they perceive such solutions to be very hard. The remainder of this tutorial describes how you can "roll your own" simulation using a GPPL. This turns out to be surprisingly easy using event graph notation. To illustrate the concepts being discussed, we have created a freely available simulation engine using the Java programming language. Java was chosen because of its platform independence and its widespread popularity, which surpassed even C and C++ in 2005 according to TIOBE Software (2006). The resulting library is surprisingly flexible, despite its tiny size.

4 REVIEW OF EVENT GRAPHS

Event graphs are a pictorial representation of event-based discrete event models. Each event in the system is represented by a vertex in the graph. State transitions can be specified below the vertex or separately, labeled by the vertex's label. Scheduling relationships between events are depicted using directed edges with attributes annotated on the edges to indicate the delay (if any) between the events and the conditions under which the scheduling should occur. When an event occurs, by convention all state transitions associated with the event are performed first. Then each edge departing from the current event's vertex is evaluated to see if its scheduling requirements are met. If so, schedule the event corresponding to the head of that edge to occur after a suitable delay. If not, take no action for this edge. If no delay is specified use a value of zero, i.e., the event being scheduled will happen at the same simulated time as the event which schedules it. If no condition is specified perform the scheduling under all circumstances.

Figure 3 illustrates the basic concepts of event graphs. **A** and **B** are events, t is a delay (which could be constant, random, or some function of the state), and **c** is a boolean function of the state. Figure 3 can be readily translated into English as follows:

> When event **A** occurs, first perform all of its state transitions. Then, if boolean condition **c** is true schedule event **B** to occur t time units later.

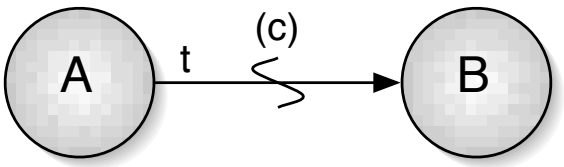

Figure 3: The Quintessential Event Graph

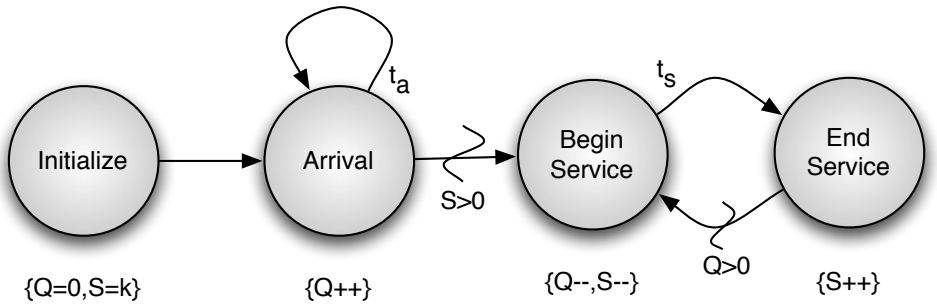

Figure 4: A G/G/k Queueing System

5 AN EVENT GRAPH EXAMPLE

Using this basic structure, we can create models of quite complex systems. For example, the model in Figure 4 represents a G/G/k queueing system (using the notation introduced in Kendall 1953). The arrival process, which instantiates values of t_a, can be anything (G $\Rightarrow$ general); the service process, which instantiates values of t_s can also be anything; and there are k servers.

By convention every event graph has an initialize event, which is scheduled to occur at the beginning of the simulation run. Initialize is used to set state variables to their initial values and to schedule such other events as are required for the model to proceed. For the G/G/k queueing system, the queue (Q) starts off with no customers and the number of servers available (S) starts at full capacity k. At least one Arrival event must occur to kick-start the model execution.

When an Arrival event occurs, the newly arrived entity is added to the queue. The next arrival is then scheduled to occur after a delay of (inter-arrival time) t_a time units. If there is a server available, a Begin Service event will be scheduled to occur immediately as well.

A Begin Service event removes one entity from the queue, and also removes one server from the available pool. It then unconditionally schedules an End Service event to occur after a delay of (service time) t_s.

When an End Service event occurs, a server is added back to the available pool. If there are entities in line, they evidently are waiting for a server to become available, so we can immediately schedule a Begin Service for the next one in the queue.

6 DESIGN CONSIDERATIONS

To implement the model as a computer program, we need three things:

- A method for each event in the event graph model that updates the model state as specified, schedules further events as appropriate, and then terminates.
- An executive loop that determines the order in which event methods should be invoked.
- A *schedule* capability that provides the mechanism by which event methods notify the executive loop about events that are candidates for invocation.

Note that only the first item is model-specific. The executive loop and scheduling capabilities are invariant across all models, and can therefore be isolated from the model implementation.

The simulation program needs to store notices of pending events in a container of some sort. Let `P` be the pending events set. A pending event notice is comprised of an event method reference and its associated time of execution. Event notices have an ordering property based on their time of execution, i.e., a notice with a smaller time should come before a notice with a larger time. We need the ability to add new event notices to `P`, and to find and extract event notice e such that $e \leq d \quad \forall d \in \{P - e\}$. A container with these capabilities is called a *priority queue*.

If `P` is implemented as a priority queue, `clock` is the simulation clock, and `current` is an event notice reference with associated method and time attributes, the following block of pseudo-code describes the structural form of the executive loop.

```
clock ← 0
invoke initialize
While ( P ≠ ∅ )
      current ← P.poll
      clock ← current.time
      invoke Current.method
```

7 IMPLEMENTATION

SIMpleKit is implemented as two classes and an interface. We will tackle these from simplest to most complex.

7.1 The Model Interface

Every SIMpleKit model must implement the `Model` interface. In Java an interface mandates class elements that

must be present, but does not provide any implementation for those elements.

The `Model` interface in SIMpleKit is small enough to present in its entirety.

```
public interface Model {
   public void initialize();
}
```

In other words, a SIMpleKit `Model` must have a method `initialize()` which takes no arguments and returns nothing.

7.2 The EventNotice Class

The `EventNotice` class is a helper class for storing and retrieving event notices. It should never be used directly by users, so all methods and constructors are declared `protected`, meaning they are not available outside of the SIMpleKit package hierarchy. Each event notice consists of a reference to an event method, the simulated time at which that event method should be invoked, and an array of Java Object's which will be used as method arguments. The method argument array can be `null` in the event that the event method takes no arguments.

Event notices are immutable. They can be created and destroyed, or can be polled via methods `time()`, `event()`, and `args()`, to retrieve the corresponding information stored by the event notice. However, those values cannot be changed once initialized by the constructor.

Finally, event notices implement Java's `Comparable` interface. This means that they implement a method `compareTo()` which provides relative ordering information for any pair of `EventNotice` objects. `EventNotice`'s are ordered by their respective times of execution.

7.3 The Simulation Class

Class `Simulation` provides the core functionality of SIMpleKit. It maintains a priority queue of `EventNotice`'s called `eventList`, a reference of type `Model` to your model object called `model`, and the current simulated time in a variable called `modelTime`. The simulated time can be polled from within a model via method `time()`.

Most of the interaction between the user's model and the `Simulation` class is handled by the `schedule` method. The method is "overloaded," i.e., there are two variants which take different arguments:

```
public static void
  schedule(String method, double delay)
```

and

```
public static void
   schedule(String method, double delay,
            Object[] args).
```

Method `schedule` takes arguments consisting of the name of the event method to be scheduled, the amount of time which should elapse before that event executes, and (optionally) an Object array containing the set of arguments to be passed to the method upon invocation. Both variants will create an `EventNotice` object and place it on the event list. If the model does not have a method which matches the specified name and argument list, an exception is thrown.

Execution of the model is handled via a method called `run()`, shown in Figure 5. The `run()` method takes one argument—a reference to your model object. The method declares a local `EventNotice` variable to keep track of the current event, sets its model reference to the model currently being run (provided as the argument to `run`, sets the simulation clock to zero, and instantiates an empty event list. Next, it invokes the `initialize()` method whose existence is guaranteed by the `Model` interface. It then proceeds with an execution loop which determines the next event to be performed by polling the event list, sets the simulation clock to the time of that event, and invokes the event method. The execution loop terminates when and if the event list is empty. Note the correspondence with the pseudo-code description of the executive loop provided in Section 6.

The last method provided by the `Simulation` class is `halt()`. This will terminate the simulation by emptying out the event list when invoked. Note that event methods should never perform scheduling activities after invoking `Simulation.halt()`. Doing so would negate the terminating condition.

8 THE M/M/k QUEUE IN SIMpleKit

We demonstrate usage of SIMpleKit by implementing an M/M/k queueing system. We can use the event graph from Section 5, since the M/M/k system is a specialization of the G/G/k in which the distributions of inter-arrival times and service times are both exponential. Choices of particular distributions do not change the event structure in any way, and can be abstracted as a call to a generator method for the desired distribution. SIMpleKit does not provide a library of distributions—techniques for random variate generation are widely available in the simulation literature.

Every SIMpleKit model must implement the `Model` interface, as described in Section 7.1. This is accomplished quite simply in the class declaration of the model.

```
public class MMk implements Model {...}
```

Looking at the state transitions and edge conditions, we note that there are only two state variables required for the model.

```
public static void run(Model m) {
    EventNotice current;
    model = m;
    modelTime = 0.0;
    eventList = new PriorityQueue<EventNotice>();
    model.initialize();
    while ((current = eventList.poll()) != null) {
        modelTime = current.time();
        try {
            current.event().invoke(model, current.args());
        } catch (Exception e) {
            e.printStackTrace(System.err);
        }
    }
}
```

Figure 5: The Simulation `run()` Method

```
// model state
private int numAvailableServers;
private int qLength;
```

Note that we are not declaring state variables to be static—you can create multiple MMk instances with distinct (or identical) parameterizations. We declare a few more instance variables to represent the parameterizations

```
// model parameters
private int maxServers;
private double arrivalRate;
private double serviceRate;
```

and initialize them with a suitable constructor.

```
public MMk(double arrivalRate,
  double serviceRate, int maxServers) {
    this.maxServers = maxServers;
    this.arrivalRate = arrivalRate;
    this.serviceRate = serviceRate;
}
```

Last but not least, we need a source of random numbers

```
// helper vars
private static Random r = new Random();
```

and a way to generate exponential random variates.

```
public double exponential(double rate) {
   return -Math.log(r.nextDouble())
          / rate;
}
```

Note that `r` is declared `static` and instantiated once, to ensure that there are no problems with overlapping sequences of random numbers in the model.

8.1 Event Methods

Each event in Figure 4 has a corresponding event method in the model class. The event graph provides a roadmap of how to write the event methods, as described in Section 4. First, perform all state transitions. Then schedule events corresponding to each departing edge if the edge conditions are `true`, with suitable delays.

The first event, `initialize()`, fulfills our contractual obligation from the `Model` interface.

```
public void initialize() {
    numAvailableServers = maxServers;
    qLength = 0;
    Simulation.schedule("arrival", 0.0);
    Simulation.schedule("halt", 100.0);
}
```

The state transitions are self-explanatory. Event scheduling is handled by asking the `Simulation` class to schedule an event with the name "arrival" with no delay. Although Figure 4 does not explicitly represent when or how to terminate the model, we need to do so for a concrete implementation. We have arbitrarily decided to create a method "halt" and schedule it to go off after 100 time units. The halt event invokes `Simulation.halt()`, which empties the event list to guarantee termination of the model as described earlier.

```
public void halt() {
   Simulation.halt();
   // report or tally results
}
```

The other event methods are sufficiently straightforward that we present the implementations without further comment other than to note that ++ and −− are Java operators for increment and decrement, respectively.

```
public void arrival() {
    ++qLength;
    Simulation.schedule("arrival",
                exponential(arrivalRate));
    if (numAvailableServers > 0) {
        Simulation.schedule(
                "beginService", 0.0);
    }
    // report state if desired
}

public void beginService() {
    --qLength;
    --numAvailableServers;
    Simulation.schedule("endService",
                exponential(serviceRate));
    // report state if desired
}

public void endService() {
    ++numAvailableServers;
    if (qLength > 0) {
        Simulation.schedule(
                "beginService", 0.0);
    }
    // report state if desired
}
```

8.2 Running the Model

The model has to be started somehow. This is accomplished by creating a Java "main" method which creates an instance of the model and passes it to the `Simulation` class to be run. For instance,

```
public static void main(String[] args) {
   Simulation.run(new MMk(4.5, 1.0, 5));
}
```

will run a model with an arrival rate of 4.5 customers per time unit, a service rate of 1.0 customers per time unit per server, and 5 servers. Multiple runs could be accomplished by looping, and statistics could be tallied and reported across runs.

9 FINAL COMMENTS ABOUT SIMPLEKIT

SIMpleKit provides a simple and transparent mechanism to illustrate how discrete event models can be implemented. Event Graphs map very directly into SIMpleKit programs, and run very efficiently. SIMpleKit is freely available under the Free Software Foundation's LGPL licensing scheme (Free Software Foundation 2006), and can be downloaded from a Subversion repository at `<https://or.nps.edu/svn/SIMpleKit>` (user=guest, password=guest).

There are several features which SIMpleKit is lacking—event cancellation, random variate generation, and hierarchical design, to name a few—but this is by design. The intent was to keep the design of SIMpleKit minimalist and use it as a pedagogical tool for understanding discrete event scheduling. Students and users who wish to build more sophisticated models are encouraged to use a more appropriate tool such as SIGMA (Schruben and Schruben 2001) or SimKit (Buss 2005), both of which strongly influenced the author's efforts.

Despite its simplicity SIMpleKit has been used to implement some quite sophisticated models, including an analysis of joint problem solving in edge vs. hierarchical organizations and an implementation of Dijkstra's algorithm for determining shortest paths in a graph.

10 CONCLUSIONS

Many people who are new to computer simulation place undue emphasis on writing the simulation program. In fact, the difficult part of a simulation study is modeling, not programming. Type III errors are all too common, and are costly both in terms of wasted time and effort and in terms of incorrect inferences or conclusions regarding the real system being modeled. Similarly, biting off more than you can chew by starting with a model which is too large or too detailed at the outset can waste time and effort. Too many studies have run out of time or budget before they even got a functioning model. Writing a good simulation program is important, but cannot possibly succeed without a good model at the core.

Keep your eyes firmly on the goal of your analysis. What is it you wish to know about the real system of interest? What are the essential characteristics and behaviors that allow you to answer your questions? Don't confuse large volumes of detail with accuracy in building your model. Start small, and add detail when and if validation shows a need for it. Test your model frequently during development, and focus on model elements which yield meaningful gains in model accuracy. These are modeling principles which apply regardless of whether you use a process or event world view, or commercial simulation packages or a GPPL.

Modern commercially available simulation software is of very high quality, and offers tremendous leverage for many problem domains. However, if you find that you're spending all of your effort trying to "trick" the software into behaving the way you want it to, consider the possibility that a different implementation approach may be more productive. Perhaps a different simulation package is more suitable for your problem. GPPLs also represent an option for your consideration. With the right tools it is surprisingly easy to implement discrete event models in a GPPL, and doing so gives you complete control over your model.

REFERENCES

Banks, J., J. S. Carson, B. L. Nelson, and D. M. Nicol. 2005. *Discrete-event system simulation*. 4^{th} ed. Upper Saddle River, N.J.: Prentice-Hall.

Brainerd, W. S., and L. H. Landweber. 1974. *Theory of computation*. Wiley.

Buss, A. H. 2005. Simkit. <http://diana.cs.nps.navy.mil/simkit>.

de Solla Price, D. 1959, June. An ancient Greek computer. *Scientific American* 60–67.

Free Software Foundation. 2006. <http://www.fsf.org/>. Free Software Foundation.

Kendall, D. G. 1953. Stochastic processes occurring in the theory of queues and their analysis by the method of imbedded Markov chains. *Annals of Mathematical Statistics* 24:338–354.

Law, A. M., and W. D. Kelton. 2000. *Simulation modeling and analysis*. 3^{rd} ed. New York, NY: McGraw-Hill.

Mernik, M., J. Heering, and A. M. Sloane. 2005. When and how to develop domain-specific languages. *ACM Computing Surveys* 37 (4): 316–344.

Mitroff, I. I., and T. R. Featheringham. 1974, November. On systemic problem solving and the error of the third kind. *Behavioral Science* 19 (6): 383–393.

Nance, R. E. 1981. The time and state relationships in simulation modeling. *Communications of the ACM* 24 (4): 173–179.

Nise, N. S. 2004. *Control systems engineering*. 4^{th} ed. John Wiley & Sons, Inc.

Sargent, R. G. 2003. Verification and validation: verification and validation of simulation models. In *Proceedings of the Winter Simulation Conference*, ed. S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, 37–48.

Schruben, D., and L. W. Schruben. 2001. *Graphical simulation modeling using SIGMA*. 4^{th} ed. Berkeley, California: Custom Simulations.

Schruben, L. W. 1983. Simulation modeling with event graphs. *Communications of the ACM* 26 (11): 957–963.

TIOBE Software. 2006, July. TIOBE Programming Community Index. <http://www.tiobe.com/index.htm?tiobe_index>.

Weinberg, G. M. 2001. *An introduction to general systems thinking*. Dorset House.

AUTHOR BIOGRAPHY

PAUL J. SÁNCHEZ is a faculty member in the Operations Research Department at the Naval Postgraduate School. His research interests include all aspects of discrete event simulation, but particularly large-scale designs of experiments. He rides recumbent bikes and has read entirely too much science fiction. You can reach him by e-mail at <PaulSanchez@nps.edu>.