# FUNDAMENTALS OF 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 random number and random variate generation, we introduce event graph notation as a simple tool for building discrete-event models. Once the model is built, we view it as a black-box which transforms inputs to outputs. This helps frame the need for designed experiments to help us gain better understanding of the system being modeled.

## 1 BACKGROUND & TERMINOLOGY

We use models in an attempt to gain understanding and insights about some aspect of the real world. There are many excellent resources available for those who wish to study the topic of modeling in greater depth than we do in this tutorial. See, for example, Law and Kelton (2000), Banks et al. (2005), Weinberg (2001), or Nise (2004).

Attempts to model reality assume *a priori* the existence of some type of "ground truth," which impartial and omniscient observers would agree upon. A first step towards success in modeling is to narrow the focus to a reasonable scope. There is a much greater chance of success for both building a model and finding consensus about the model's utility when we focus our attention locally in time and space. We will start our study of models at the level of a system.

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 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 shear during landing on a flight simulator than by going out and practicing real landings in wind shear 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 shear 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 that are comprised of model entities similar to those in the real system, and that 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.

One huge assumption we make when modeling a system is *observability*, i.e., we assume that by observing the system for a sufficiently long time we can infer the state and quantify the relationships between inputs and outputs. Mathematical systems theory (Nise 2004, Weinberg 2001) shows that this assumption is not a given. Linear systems are amongst the simplest of systems, yet even some linear systems can be proven to be non-observable. However, without the assumption of observability there's no way to proceed. If the intended use of the model is to "tune" the system, we should also be concerned about the related issue of *controllability*, i.e., can the system be driven from its current state to the desired state in finite time using finite inputs?

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 space. 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 Canada geese that 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

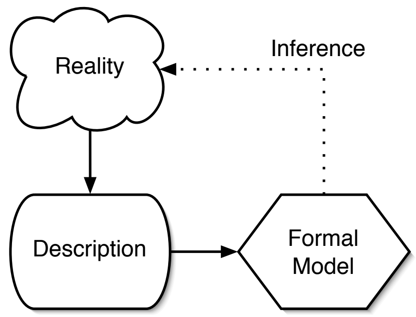

Figure 1: A Model Yields Insights and Inferences

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

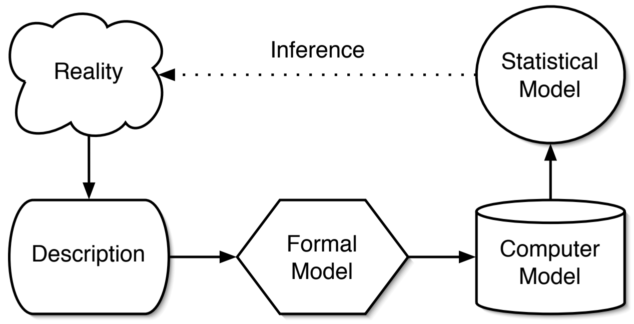

Figure 2: Simulation Has a Longer Chain of Inference

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 Figure 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 many people advocate using analytical solutions where possible. It's often the case, however, that modelers make unrealistic assumptions in order to attain analytical tractability. The conclusions you derive from such a model may be completely inappropriate for the real-world system being modeled because of the distortions added for tractability. See Lucas et al. (2015) for more details.

In addition to modeling real-world phenomena that analytical models can't, 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 data because you have total control over generating them.

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?

- **Backtrack / 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

Systems can be classified in a variety of ways. We will 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 is a concept identified as a *Type III error* by Mitroff and Featheringham (1974). They defined it as "the error…[of] choosing the wrong problem representation…" This can happen, for example, 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). Informally, any computing device, programming system, or language which is capable of processing any computationally feasible problem is said to be *Turing complete* (Brainerd and Landweber 1974). Turing completeness is a prerequisite for a language to be considered a GPPL. By definition Turing complete languages are capable of supporting simulation modeling, but 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. It turns out to be surprisingly easy with event graphs, a graphical modeling notation which we will introduce in Section 5. First, though, we need to explore the topic of randomness in computer programs.

## 4 RANDOMNESS

Randomness plays an essential role in simulation modeling. It is an integral part of many systems which directly affects their behaviors. It can also be used to reflect our uncertainty about the details of real-world behaviors—in many cases, distribution modeling can be used in place of detailed physical models for complex subsystems.

### 4.1 The Importance of Randomness in Models

Let's do a small thought experiment. Consider a production line in which component pieces A and B are delivered to a robotic arm at precise one minute intervals. The robot assembles and welds those components, taking exactly one minute to do so. It should be clear after a moment's thought that if we started with no queue of components, that will not change. Similarly, if we started with a queue of N of each component, we will vary from N by no more than one (depending on the synchronization of times of arrival and time at which the robot cycles to the next assembly). In a queueing system with purely deterministic behavior, there is no buildup of the line as long as the arrival rate is less than or equal to the service rate.

Now consider what happens if the robot breaks down occasionally or the time between component arrivals varies. Queueing theory tells us that if the arrival rate is greater than *or equal to* the service rate, the queue lengths will grow in an unbounded fashion.

The two systems in our thought experiment behave in radically different fashions, yet the only difference between them is whether the arrival and/or service times are deterministic or random. In other words, system behavior can be drastically affected by the presence or absence of randomness. Many beginning modelers are tempted to simplify their model by avoiding randomness. They almost invariably do so by replacing random variables with their means. This can dramatically change the behavior of the model.

## 4.2 Random Numbers and Random Variates

Simulationists distinguish between *random numbers*, which are uniformly distributed between zero and one ($U(0,1)$), and *random variates*, which are everything else. Given a source of random numbers, in principle we can generate random variates with any distribution desired (see Appendix A). In practice random variates are generated using a broad variety of techniques, as illustrated in Section 4.3. Much more information can be found in works such as Banks et al. (2005), Devroye (1986), Law and Kelton (2000), or Leemis and Taber (2007).

Randomness is an essential component of many models, so we need a source of randomness that we can draw on within our simulation program. Let's consider what would constitute desirable characteristics for such a source. For each item in the following wish list, we briefly describe why it is desirable.

- **Independence** – many experiments require independent observations. It's much easier to induce correlation from independent random numbers than to achieve independence from correlated ones.
- **Known distribution** – we would prefer $U(0,1)$'s, but using the results in Appendix A we can, in principle, map any known distribution to uniform. From there we can in turn map it to other distributions.
- **Unlimited** – we should be able to draw as many values from the source as we need, with no prior limitations.
- **Minimal storage** – the source should not be a drain on memory or other storage resources.
- **Fast** – we often need lots of random numbers, so getting them should be quick.
- **Portable** – our ability to create and run models should not be tied to a specific computing platform.
- **Reproducible** – we need to be able to repeat the sequence of values for debugging purposes, so colleagues can confirm our results, and for use in covariance induction strategies ( often referred to as "variance reduction techniques").

Obviously some of these items are mutually incompatible. Several solutions have been tried over the years.

At first glance, trace-driven simulation seems appealing. That is where historical data are used directly as inputs. It's hard to argue about the validity of the distributions when real data from the real-world system is used in your model. In practice, though, this tends to be a poor solution for several reasons. Historical data may be expensive or impossible to extract. It certainly won't be available in unlimited quantities, which significantly curtails the statistical analysis possible. Storage requirements are high. And last, but not least, it is impossible to assess "what-if?" strategies or try to simulate a prospective system, i.e., one which doesn't yet exist.

At the other end of the spectrum, people have built physical collection devices such as geiger or cosmic ray counters, thermal noise sensors, quantum photoelectric effects, and even optical sensors monitoring lava lamps. Drawbacks to using hardware based randomness include unknown distributions and dependence structures, speed, portability, and reproducibility.

The most widespread solution is the use of Pseudo-Random Number Generators (PRNGs), which use algorithms to transform internal state information into a sequence of output values. So how do we achieve randomness with PRNGs? The answer is, we don't. PRNGs are computer programs, and given the same inputs they will produce the same output every time. They most definitely are reproducible, and most definitely are not independent. On the plus side, PRNGs can have provably uniform distributions, can be fast and portable, and usually have minimal storage requirements. They are not unlimited – they are based on a finite state space and will eventually cycle after enumerating all possible states. In practice this is not always a limitation. For instance, the Mersenne Twister (Matsumoto and Nishimura 1998) has a cycle length of $2^{19937}-1$. That is more than $10^{5000}$, i.e., a one followed by five thousand zeroes! If every atom in the universe was a computer capable of performing trillions of operations per second, and had started using random numbers at the creation of the universe 15 billion years ago, we still wouldn't have used a significant portion of a $2^{19937}-1$ cycle.

We know the outcomes of a PRNG are not random, since they are arithmetical and can be repeated on demand. So how do we justify using them? To answer the question let's consider the "imitation game" proposed by Turing (1950), and better known now as the Turing test. Roughly, the Turing test proposed setting up two teletype devices, one with a human at the other end and the other with a computing device. If, with unbounded interaction, you couldn't tell which was which, then the machine could be viewed as an intelligent actor. We apply the Turing test in our context by replacing the human and the computer with a source of "true" randomness and a PRNG. If you're allowed to apply any test you like and can't distinguish between the two, then we consider the PRNG to be a suitable substitute for the "true" randomness. Comprehensive suites of statistical tests have been developed and are freely available (Marsaglia 1995, Brown 2004).

You should never try to create your own PRNG unless you have invested years of study in advanced mathematics and computational numerical methods. Random number generation is an extremely difficult and hazardous undertaking that stumped some of the best mathematical minds of the 20th century. Von Neumann's infamous quote, "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin" (Knuth 1981) was inspired by the difficulties he encountered in creating the "middle-square" method. The IBM Corporation also failed spectacularly with RANDU, about which Knuth (1981) said (p.104) the "... very name RANDU is enough to bring dismay into the eyes and stomachs of many computer scientists!" If you are using a GPPL to implement your simulation, use a PRNG which has been thoroughly tested such as the Mersenne Twister (Matsumoto and Nishimura 1998, Matsumoto 2007), `ranlux` (Lüscher 1994), or the combined recursive generator by L'Ecuyer (1996), all of which are freely available from the Free Software Foundation (FSF) (2007). If you are working with a commercial simulation package a PRNG will almost certainly be provided as a built-in function, but the wise modeler will confirm that it is a reputable one.

## 4.3 Some Random Variate Generation Examples

Once we have a good source of random numbers, we can use them to produce random variates using a variety of techniques. Throughout this section we will use $R$ to denote a TRI(0,1,0) distribution, i.e., a right-triangular distribution with the min and mode both zero and a max of one; and $X$ to denote a TRI(-1,1,0) distribution, i.e., a symmetric triangle with a min and max of minus one and one, respectively, and the mode at zero.

### 4.3.1 Inversion

To generate values from $R$ using inversion, we need to derive the CDF. The density function is straightforward:

$$f(r) = \begin{cases} 2(1-r) & 0 \le r \le 1 \\ 0 & \text{otherwise,} \end{cases}$$

and the CDF is therefore

$$F_R(r) = \begin{cases} 0 & r < 0 \\ 2r - r^2 & 0 \le r \le 1 \\ 1 & r > 1. \end{cases}$$

After adding and subtracting 1 to complete the square, we can re-express this as

$$F_R(r) = \begin{cases} 0 & r < 0 \\ 1-(1-r)^2 & 0 \le r \le 1 \\ 1 & r > 1. \end{cases} \tag{1}$$

To invert, we set $F_R(R) = U$ and solve for $R$:

$$\begin{aligned} F_R(R) &= U \\ 1-(1-R)^2 &= U \\ (1-R)^2 &= 1-U \\ 1-R &= \sqrt{1-U} \\ R &= 1-\sqrt{1-U}. \end{aligned}$$

### 4.3.2 Convolution

Convolution is the fancy term for adding random variables. For example, it's well known that adding two $U(0,1)$'s yields a TRI(0,2,1) distribution. If we would like a random variate $X$ with a TRI(-1,1,0) distribution we can shift a TRI(0,2,1) to the left by subtracting 1:

$$X = U_1 + U_2 - 1.$$

### 4.3.3 Composition

Another way to generate $X$ is to note that it is symmetrically composed from two right-triangles, $-R$ and $R$, each of which makes up half of the distribution. Since we already know how to generate $R$ with inversion, it's simple to create the composition

$$X = \begin{cases} -R & \text{if } U \le 0.5 \\ R & \text{otherwise.} \end{cases}$$

### 4.3.4 Special Relationships

Suppose we are interested in the distribution of the $M$, the minimum of two independent $U(0,1)$'s. We can derive the CDF as follows:

$$\begin{aligned} F_M(m) &= P\{M \le m\} \\ &= 1 - P\{U_1 > m \ \cap \ U_2 > m\} \\ &= 1 - \prod_{i=1}^{2}(1 - P\{U_i \le m\}) \\ &= 1 - (1 - F_U(m))^2 \\ &= 1 - (1-m)^2 \end{aligned}$$

since $F_U(u) = u$ for a $U(0,1)$. Once we recognize this as identical to equation 1, the CDF of the TRI(0,1,0) distribution, we can see that generating $M$ and generating $R$ are interchangeable problems. We could generate $M$ via the inversion for $R$. Alternatively, we could generate $R$ by generating two $U$'s and selecting the minimum value. We might choose the former if we wanted to use a correlation induction strategy, or the latter if it was computationally faster.

## 5 A QUICK INTRODUCTION TO EVENT GRAPHS

Event graphs are a visual 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.

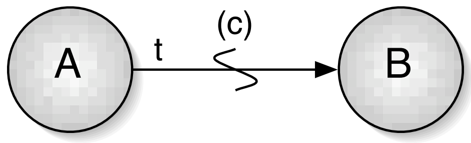

Figure 3: The Quintessential Event Graph

### 5.1 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 by 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 `init` event which is automatically scheduled to start the simulation, sets state variables to their initial values, and does preliminary scheduling of other events to "kick-start" your model. 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.

### 5.2 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 $e \leq d \quad \forall d \in \{\texttt{P} - e\}$. 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

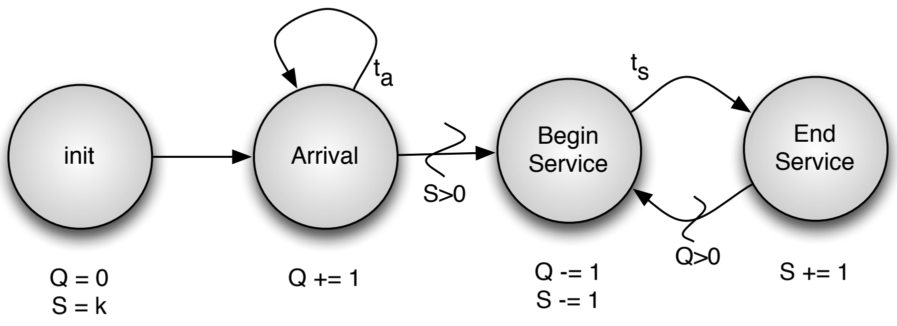

Figure 4: A G/G/k Queueing System

with associated method and time attributes, the following block of pseudo-code describes the structural form of the executive loop.

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

### 5.3 SIMpleKit-python Implementation

SIMpleKit-python is a minimalist software library for building and running event-graph models, implemented in the Python programming language (**?**) due to its widespread popularity. It offers readability, and cross-platform availability and portability.

SIMpleKit-python is implemented as an abstract class which provides a small number of methods and requires you to implement a method named `init` to start the simulation, as described in Section 5.1. To make your Python program a SIMpleKit-python model, you must:

- import the *SimpleKit* class;
- create a model class which subclasses *SimpleKit*;
- have a zero-argument method named `init` (which should schedule at least one other event method as events);
- implement event methods within your model class corresponding to the events in your event graph, which can be named anything you wish **except** `run`, `schedule`, `halt`, or `model_time`; and
- kick-start your model by sending an instance of your model class a `run` message.

Within your model you can use the `model_time` method to access the simulation clock's current value; the `schedule` method to schedule further events, with or without arguments; the `halt` method to terminate the model's execution; and the `run` method to invoke execution on an instance of your model class. All of these features will be illustrated in Section 5.4 with a working example.

#### 5.3.1 The SimpleKit Class

The entire source code for SIMpleKit is provided in Figure 5. The lettered blue circles flag particular items of interest, referred to in the text by the annotation letter.

The `SimpleKit` class is implemented as an abstract base class via the `abc` module (annotation A), which means that you cannot directly instantiate an object of type `SimpleKit`. Instead, you create a model class which is a subclass of `SimpleKit` and override the abstract `init` method (annotation B) to kick-start your model.

The `SimpleKit` class provides two instance variables (annotation C), `event_list`—a `PriorityQueue` of pending `EventNotice` objects—and `model_time`, which is self-descriptive.

`EventNotice` (annotation G) is an internal class for storing an event's method reference, associated time of execution, and priority. Event notices are created and stored in the `event_list` by the `schedule` method (annotation D), which requires two or more user-provided arguments. The first argument is the name of the event method to invoke. The second argument is the delay until the event being scheduled should be invoked. Any additional argument values will be stored in the event notice, to be included as arguments to the event method upon invocation. The `event_list` orders event notices by their time of execution with ties broken by priority, an optional final argument to the `schedule` method.

Execution of the model is handled via the `run` method (annotation E), which invokes the `init` method (which you will have overridden) to initialize your model. It then proceeds with an execution loop which determines the next event to be performed by polling the event list, updates the simulation clock to the time of that event, and invokes the event method with any arguments that may have been

```
    """A discrete event modeling toolkit based on event graphs."""

    from queue import PriorityQueue
    import abc

    class SimpleKit:
        """
        To create a SimpleKit model:
            Your model class must be a subclass of SimpleKit.
            Your constructor (__init__()) must call SimpleKit.__init__(self).
            You must override the abstract init() method to start your model.
        Note:
            Your model cannot implement methods named run(), schedule(), or halt().
        """
A       __metaclass__ = abc.ABCMeta

        def __init__(self):
C           """Initialize a pending events list and set model_time to 0."""
            self.event_list = PriorityQueue()
            self.model_time = 0.0

        @abc.abstractmethod
B       def init(self):
            """This abstract method must be overridden in your model class."""

        def run(self):
E           """Execute the model logic."""
            self.init()
            while not self.event_list.empty():
                event_notice = self.event_list.get_nowait()
                self.model_time = event_notice.time
                event_notice.event(*(event_notice.args))

        def schedule(self, event, delay, *args, priority=10):
D           """Add an event to the pending events.

            Args:
                event: The name of the event method to be scheduled.
                delay: The amount of model time by which to delay the execution.
                args: (optional) Any arguments required by the event.
            """
            if delay < 0:
                raise RuntimeError('Negative delay is not allowed.')
            self.event_list.put_nowait(
                self.EventNotice(event, delay + self.model_time, args, priority))

        def halt(self):
F           """Terminate a simulation run by clearing the pending events list."""
            while not self.event_list.empty():
                self.event_list.get_nowait()

        class EventNotice:
G           """Internal class for storage & retrieval of event notice info."""

            def __init__(self, event, time, args, priority=10):
                self.event = event
                self.time = time
                self.priority = priority
                self.args = args

            def __eq__(self, other):
                return self.time == other.time and self.priority == other.priority

            def __lt__(self, other):
                return (self.time < other.time or
                    (self.time == other.time and self.priority < other.priority))
```

Figure 5: SIMpleKit-python Source

provided. The execution loop terminates when and if the event list becomes empty. Note the almost one-to-one correspondence with the pseudo-code description of the executive loop provided in Section 5.2.

The last method provided by the `SimpleKit` class is `halt` (annotation F). This will terminate the simulation by emptying out the event list when invoked, which in turn will halt the loop in the `run` method. Consequently, note that event methods should never perform scheduling activities after invoking `halt`. Doing so would negate the terminating condition.

## 5.4 The M/M/k Queue in SIMpleKit

We will demonstrate usage of SIMpleKit-python by implementing an M/M/k queueing system. The resulting program is presented in its entirety as Figure 6. The numbered blue circles flag particular items of interest, referred to in the text by the annotation number. The implementation is based on the event graph in Figure 4, since the M/M/k system is a realization of a 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-python does not implement random variate generation. Users are encouraged to use `numpy`'s distributional generators, or utilize any of the variety of techniques for random variate generation that are widely available in the simulation literature. We leave it to the model builder to choose a solution suitable for their specific needs.

We start by importing the `SimpleKit` class and the `numpy` module for random variate generation (annotation 1). The model's class name is `MMk`, and it subclasses `SimpleKit` (annotation 2).

With object-oriented modeling you can create as many MMk instances as you wish with distinct (or identical) parameterizations. Individual model parameterization is established via the class constructor `__init__` (annotation 3), which sets the arrival and service rates as well as the number of servers. Rates are then converted to their equivalent means by inverting, since `numpy`'s exponential generator parameterizes the distribution using mean.

The utility method `dumpState` (annotation 4) is used to print a snapshot of the model state after each event.

### 5.4.1 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 5.2. First, perform all state transitions. Then schedule events corresponding to each departing edge if the edge conditions are `true`, with suitable delays.

The various event implementations are found starting at annotation 5. The first event, `init`, is required to override the abstract version found in `SimpleKit` and provide an entry point to the model. The state transitions are self-explanatory. The first call to `schedule` will create and store an event notice for an `arrival` event, with no delay and no additional arguments. Although Figure 4 does not explicitly designate when or how to terminate the model, in practice we need some stopping criterion for a concrete mplementation. We have arbitrarily decided to schedule the `halt` method (from the `SimpleKit` parent class) to occur after a delay of 100 time units. This is done by the second call to `schedule`. Finally, there is a call to `dumpState` at the end of `init` and each of the event methods to provide a trace of the model's behavior.

The other event methods are sufficiently straightforward that we leave them for your inspection without further comment, other than to note that += and −= are Python's increment-by and decrement-by operators, respectively.

### 5.4.2 Running the Model

After defining the model class, it is straightforward to instantiate, parameterize, and execute the model (annotation 6). The command

```
MMk.new(4.5, 1.0, 5).run()
```

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

## 5.5 Final Comments About SIMpleKit

SIMpleKit-python is intended to illustrate how discrete event models can be implemented in general purpose programming languages. Event Graphs map very directly into SIMpleKit-python programs, and run very efficiently.

The software shown here is freely available under the **?** LGPL license. It can be downloaded using the **?** command

```
git clone \
     https://gitlab.nps.edu/pjsanche/simplekit-python.git
```

where \ indicates continuation on the same line.

SIMpleKit-python is lacking several features which are needed for more advanced modeling—event cancellation and hierarchical design, for instance—but this is by design. The intent was to minimize the footprint of `SimpleKit` so it can be used as a pedagogical tool for understanding discrete event modeling. Users have access to all of Python's extensive libraries for random variate generation, statistics, and graphics, so SIMpleKit-python focuses on the event scheduling mechanism. Those who wish to build more sophisticated models are encouraged to use a more

```
    """Demo model of SimpleKit usage."""
    from simplekit import SimpleKit
1   import numpy
    import math

2   class MMk(SimpleKit):
        """Implementation of an M/M/k queueing model using SimpleKit."""

3       def __init__(self, arrivalRate, serviceRate, maxServers):
            """Construct an instance of the M/M/k."""
            SimpleKit.__init__(self)
            self.meanArrival = 1.0 / arrivalRate
            self.meanSvc = 1.0 / serviceRate
            self.maxServers = maxServers
            self.qLength = 0
            self.numAvailableServers = 0

5       def init(self):
            """Initialize all state variables, schedule first arrival and halt."""
            self.numAvailableServers = self.maxServers
            self.qLength = 0
            self.schedule(self.arrival, 0.0)
            self.schedule(self.halt, 100.0, priority = 0)
            self.dumpState("Init")

        def arrival(self):
            """Increment queue, schedule next arrival, beginService if possible."""
            self.qLength += 1
            self.schedule(self.arrival, numpy.random.exponential(self.meanArrival))
            if self.numAvailableServers > 0:
                self.schedule(self.beginService, 0.0, priority = 2)
            self.dumpState("Arrival")

        def beginService(self):
            """Remove customer from line, allocate server, schedule endService."""
            self.qLength -= 1
            self.numAvailableServers -= 1
            self.schedule(self.endService, numpy.random.exponential(self.meanSvc))
            self.dumpState("beginService")

        def endService(self):
            """Free server, if customers are waiting initiate another service."""
            self.numAvailableServers += 1
            if self.qLength > 0:
                self.schedule(self.beginService, 0.0, priority = 1)
            self.dumpState("endService")

4       def dumpState(self, event):
            """Dump of the current state of the model."""
            print("Time: %6.2f" % self.model_time, "  Event: %-12s" % event,
                  "  Queue Length: %3d" % self.qLength, " Available Servers: ",
                  self.numAvailableServers)

6   if __name__ == '__main__':
        numpy.random.seed(12345)
        MMk(4.5, 1.0, 5).run()      # Instantiate and run a copy of the MMk model.
```

Figure 6: Annotated Source Code for the M/M/k Queueing Model

appropriate tool such as SimKit (Buss 2005), which strongly influenced the design of SIMpleKit-python.

Despite its minimalist structure, implementations of SIMpleKit have been used to build a variety of quite sophisticated programs. Examples include stochastic Lanchester force-on-force combat; how joint problem solving is accomplished in edge vs. hierarchical organizations; and an implementation of Dijkstra's algorithm for determining shortest paths in a graph.

## 6 VIEWING YOUR MODEL AS A MODEL

Let's now step back from the internal details of your model and view it as a parameterized "black box," as in Figure 7. From that perspective your model, like other systems, transforms inputs into outputs. The inputs are the parameterizations of each run of the model, e.g., for an M/M/k queueing model the arrival rate, service rate, and number of servers. However, we can take a broader view of what constitutes a simulation input. Inputs also include the random number seeds if, for instance, we wish to use common random numbers for variance reduction.

Extending our view even further, we can consider an M/M/k system to be a particular case of a G/G/k system. In that case, the choice of exponential distribution is a parameterization, and the impact of distribution choices could be explored. If this seems to be more trouble than it's worth, consider that explorations of distribution and parameter choices can be done *before* you do significant input distribution modeling. The results can help you focus your scarce resources on those choices which matter. If your simulation is robust to the distribution of a particular component element, use a distribution which is easy to implement and computationally efficient. On the other hand, if your simulation results vary significantly based on the choice of distribution, this is a model component which is important for you to expend the effort to get right.

This black box view of our model highlights the potential for design of experiments to greatly enhance our understanding of the model. Please see Sanchez (2007) for an excellent introduction to these concepts.

## 7 OUTPUT ANALYSIS

Once you have built your simulation model, it's time to make it work for you. That means analyzing its input/output behavior to try to gain insights about the model, and by inference about the real-world system being modeled. The nature of your model should determine the type of analysis you use. The choices are represented in Figure 8 as a decision tree. The term *classical statistics* will be used to describe the broad variety of statistical techniques which assume independent observations.

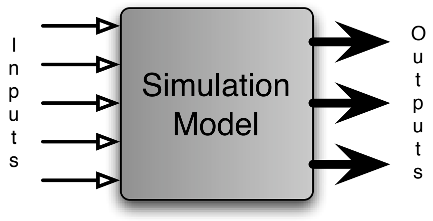

Figure 7: Simulation as a Black Box

If there is no randomness in your model, you can run it once to determine "the answer." Multiple runs will give you no additional information. For the remainder of this discussion, the model is assumed to have randomness.

If your model is of a static system, or has time dependency but has a well-defined terminating state, you should use replication to study it. Each iteration of a static system, or each run of a terminating system, yields an independent observation. These data can be collected and then analyzed using classical statistics.

If you have a dynamic non-terminating simulation, the system must be studied as a *time series* (Chatfield 1996, Box and Jenkins 1976). If the system eventually settles into a mode where the distribution of the outputs is independent of time, it is said to be *stationary*. Note that stationary systems still vary over time, but that the outcomes cannot be predicted by knowing the time. With time series data we need to be concerned about the impact of *serial correlation*, which is a measure of the propensity for lingering effects in the system's performance over time. Serial correlation affects estimates of both the mean (via *initial bias*) and the variance of the time series—classical statistics cannot be used directly. These issues were identified by Conway (1963), and over the intervening years many researchers have developed a variety of techniques to deal with them. The simplest of these is *replication/deletion*. In replication/deletion, you first delete some initial portion of the simulation output to remove the effects of initialization. Averaging the remaining data yields an unbiased estimate of the steady-state mean behavior. You can then use replication to obtain a suitable number of these estimates, which will be independent if the replications are independently seeded. The results are independent and identically distributed observations which can be analyzed with classical statistics. For more information, see any modern simulation textbook (Law and Kelton 2000, Banks et al. 2005).

If your system is non-stationary, is it because of cyclic or trend variations in the mean? If so, it may be possible to explicitly fit a harmonic or asymptotic model of the mean. The residuals from that model could then be analyzed as covariance stationary data. However, such an analysis is well beyond the scope of this introductory text.

If none of the above apply, we have run out of options. I wish you the best of fortune, and look forward to hearing how you dealt with the problem.

## 8 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

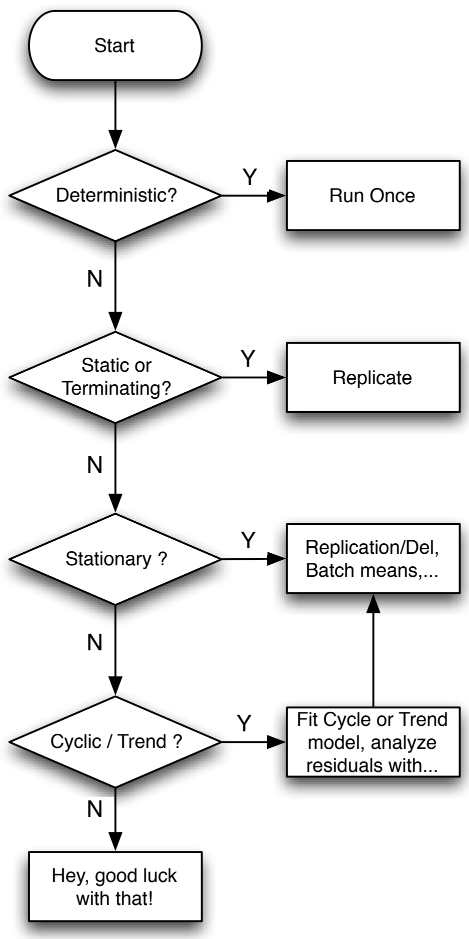
Start
Deterministic?
Y
Run Once
N
Static or Terminating?
Y
Replicate
N
Stationary ?
Y
Replication/Del, Batch means,...
N
Cyclic / Trend ?
Y
Fit Cycle or Trend model, analyze residuals with...
N
Hey, good luck with that!

Figure 8: Select an Appropriate Analysis Methodology

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.

If you are using a commercial simulation package, make sure that it is using a reasonable quality random number generator and that you understand how to control the seeding of that generator to achieve independence or induce dependence between your runs, whichever is appropriate for your plan of analysis. If you are using a GPPL, adopt a reputable implementation rather than trying to "roll your own." Similarly, use any of the wide variety of published algorithms to generate the random variates for your simulation.

Leave sufficient time for analysis of your model. A surprising (and depressing) number of people build their model, run it once, and claim they now know "the answer." If your simulation involves randomness, you must use statistics to analyze it. The appropriate statistical analysis depends on the characteristics of your model, and using inappropriate techniques can invalidate your analysis.

Lastly, use designed experiments. These techniques should be applied not only during the final analysis phase, but also during the course of model development. They can help you focus your efforts most productively during modeling, and maximize the amount of information you can extract from your model during analysis. Design of experiments should be in every analyst's toolbox.

## A INVERSION

We assume here that the reader is familiar with basic probability theory and common notation. Recall that the definition of the Cumulative Distribution Function (CDF) of a random variable $X$ is

$$F_X(b) \equiv P\{X \le b\}.$$

Recall also that the distribution of a random variable is uniquely identified by its CDF, and that a random variable with a uniform(0,1) distribution has CDF

$$F_U(b) = \begin{cases} 0 & b < 0 \\ b & 0 \le b \le 1 \\ 1 & b > 1. \end{cases}$$

Now consider a random variable $X$ with invertible CDF $F_X$. In general a function of a random variable is itself a random variable. So what is the distribution of the random variable $Y = F_X(X)$, i.e., what do we get when we apply its own CDF to random variable $X$? The answer to that question is both surprising and extremely useful:

$$\left.\begin{aligned} F_Y(b) &= P\{Y \leq b\} \\ &= P\{F_X(X) \leq b\} \\ &= P\{X \leq F_X^{-1}(b)\} \\ &= F_X(F_X^{-1}(b)) \\ &= b \end{aligned}\right\} \qquad 0 \leq b \leq 1.$$

In other words, $F_X(X)$ has a $U(0,1)$ distribution, regardless of the distribution of $X$! If we have a source for $U(0,1)$ random numbers, we can in principle convert them into random variates with distribution $F_X()$ as follows:

$$F_X(X) = U \quad \Longrightarrow \quad X = F_X^{-1}(U).$$

## REFERENCES

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

Box, G. E. P., and G. M. Jenkins. 1976. *Time series analysis: forecasting and control*. Holden-Day.

Brainerd, W. S., and L. H. Landweber. 1974. *Theory of computation*. New York, NY: John Wiley & Sons, Inc.

Brown, R. G. 2004. Robert G. Brown's General Tools Page. <http://www.phy.duke.edu/~rgb/General/dieharder.php>.

Buss, A. H. 2005. SimKit. <https://eos.nps.edu/Simkit/>.

Chatfield, C. 1996. *The analysis of time series: an introduction*. 5th ed. Chapman & Hall/CRC.

Conway, R. W. 1963, October. Some tactical problems in digital simulation. *Management Science* 10 (1): 47–61.

de Solla Price, D. 1959, June. An ancient Greek computer. *Scientific American* 200 (6): 60–67.

Devroye, L. 1986. *Non-uniform random variate generation*. New York, NY: Springer-Verlag.

Free Software Foundation (FSF) 2007. GSL – GNU Scientific Library. <http://www.gnu.org/software/gsl/>.

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.

Knuth, D. E. 1981. *The art of computer programming: Seminumerical algorithms*. Second ed, Volume 2. Reading, MA: Addison-Wesley Publishing Company.

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

L'Ecuyer, P. 1996. Combined Multiple Recursive Random Number Generators. *Operations Research* 44 (5): 816–822.

Leemis, L. M., and J. G. Taber. 2007. Univariate distribution relationships. Technical report, The College of William & Mary, Department of Mathematics.

Lucas, T. W., W. D. Kelton, P. J. Sánchez, S. M. Sanchez, and B. L. Anderson. 2015. Changing the paradigm: Simulation, now a method of first resort. *Naval Research Logistics* 62 (4): 293–305.

Lüscher, M. 1994. A portable high-quality random number generator for lattice field theory calculations. *Computer Physics Communications* 79:100–110.

Marsaglia, G. 1995. The Marsaglia Random Number CDROM including the Diehard Battery of Tests. <http://stat.fsu.edu/pub/diehard/>.

Matsumoto, M. 2007. Mersenne Twister: A random number generator (since 1997/10). <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>.

Matsumoto, M., and T. Nishimura. 1998, Jan.. Mersenne Twister: A 634-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transsactions on Modeling and Computer Simulation* 8 (1): 3–30.

Mernik, M., J. Heering, and A. M. Sloane. 2005. When and how to develop domain-specific languages. *ACM Comput. Surv.* 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. *Commun. ACM* 24 (4): 173–179.

Nise, N. S. 2004. *Control systems engineering*. 4th ed. New York, NY: John Wiley & Sons, Inc.

Open Source Initiative 2019. The MIT license. <https://opensource.org/licenses/MIT>.

Ruby. <http://www.ruby-lang.org/en/>.

Sanchez, S. M. 2007. Work Smarter, Not Harder: Guidelines for Designing Simulation Experiments. In *Proceedings of the 2007 Winter Simulation Conference*, ed. S. G. Henderson, B. Biller, M.-H. Hsieh, J. Shortle, J. D. Tew, and R. R. Barton. Piscataway, NJ: Winter Simulation Conference: IEEE.

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

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

Turing, A. 1950, October. Computing machinery and intelligence. *Mind* LIX (236): 433–460.

Weinberg, G. M. 2001. *An introduction to general systems thinking*. New York, NY: Dorset House Publishing Co., Inc.