

This book is the basis for a first course on *discrete-event simulation*. That is, the book provides an introduction to computational and mathematical techniques for *modeling, simulating* and *analyzing* the performance of discrete-event stochastic systems. By definition, the nature of discrete-event simulation is that one does not actually experiment with or modify an actual system. Instead, one develops and then works with a discrete-event simulation *model*. Consistent with that observation, the emphasis in this first chapter is on model building.

1.1.1 MODEL CHARACTERIZATION

Briefly, a discrete-event simulation model is both stochastic and dynamic with the special discrete-event property that the system state variables change value at discrete times only (see Definition 1.1.1). But what does that mean?

A system model is *deterministic* or *stochastic*. A deterministic system model has no stochastic (random) components. For example, provided the conveyor belt and machine never fail, a model of a constant velocity conveyor belt feeding parts to a machine with a constant service time is deterministic. At some level of detail, however, all systems have some stochastic components; machines fail, people are not robots, service requests occur at random, etc. An attractive feature of discrete-event simulation is that stochastic components can be accommodated, usually without a dramatic increase in the complexity of the system model at the computational level.

A system model is *static* or *dynamic*. A static system model is one in which time is not a significant variable. For example, if three million people play the state lottery this week, what is the probability that there will be at least one winner? A simulation program written to answer this question should be based on a static model; when during the week these three million people place their bets is not significant. If, however, we are interested in the probability of no winners in the next four weeks, then this model needs to be dynamic. That is, experience has revealed that each week there are no winners, the number of players in the following week increases (because the pot grows). When this happens, a dynamic system model must be used because the probability of at least one winner will increase as the number of players increases. The passage of time always plays a significant role in dynamic models.

A dynamic system model is *continuous* or *discrete*. Most of the traditional dynamic systems studied in classical mechanics have state variables that evolve continuously. A particle moving in a gravitational field, an oscillating pendulum, or a block sliding on an inclined plane are examples. In each of these cases the motion is characterized by one or more differential equations which model the continuous time evolution of the system. In contrast, the kinds of queuing, machine repair and inventory systems studied in this book are discrete because the state of the system is a *piecewise-constant* function of time. For example, the number of jobs in a queuing system is a natural state variable that only changes value at those discrete times when a job arrives (to be served) or departs (after being served).

The characterization of a system model can be summarized by a tree diagram that starts at the system model root and steps left or right at each of the three levels, as illustrated in Figure 1.1.1.

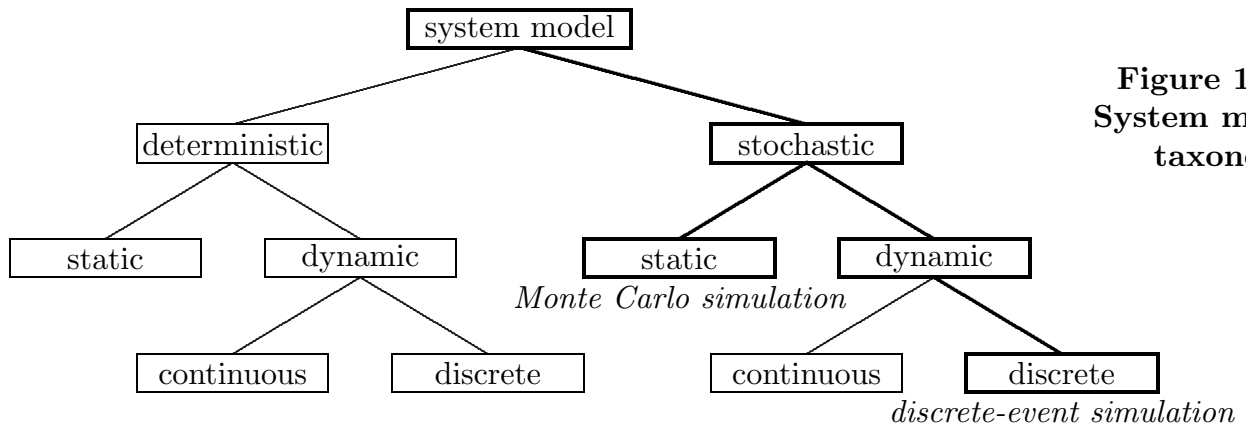


Figure 1.1.1.
System model
taxonomy.

As summarized by Definition 1.1.1, the system model characterized by the right-most branch of this tree is of primary interest in this book.

Definition 1.1.1 A *discrete-event simulation model* is defined by three attributes:

- *stochastic* — at least some of the system state variables are random;
- *dynamic* — the time evolution of the system state variables is important;
- *discrete-event* — significant changes in the system state variables are associated with events that occur at discrete time instances only.

One of the other five branches of the system model tree is of significant, but secondary, interest in this book. A *Monte Carlo simulation model* is stochastic and static — at least some of the system state variables are random, but the time evolution (if any) of the system state variables is not important. Accordingly, the issue of whether time flows continuously or discretely is not relevant.

Because of space constraints, the remaining four branches of the system model tree are not considered. That is, there is no material about deterministic systems, static or dynamic, or about stochastic dynamic systems that evolve continuously in time.

1.1.2 MODEL DEVELOPMENT

It is naive to think that the process of developing a discrete-event simulation model can be reduced to a simple sequential algorithm. As an instructional device, however, it is useful to consider two algorithms that outline, at a high level, how to develop a discrete-event simulation model (Algorithm 1.1.1) and then conduct a discrete-event simulation study (Algorithm 1.1.2).

Algorithm 1.1.1 If done well, a typical discrete-event simulation model will be developed consistent with the following six steps. Steps (2) through (6) are typically iterated, perhaps many times, until a (hopefully) valid computational model, a computer program, has been developed.

- (1) Determine the *goals* and *objectives* of the analysis once a system of interest has been identified. These goals and objectives are often phrased as simple Boolean decisions (e.g., should an additional queuing network service node be added) or numeric decisions (e.g., how many parallel servers are necessary to provide satisfactory performance in a multi-server queuing system). Without specific goals and objectives, the remaining steps lack meaning.
- (2) Build a *conceptual* model of the system based on (1). What are the *state variables*, how are they interrelated and to what extent are they dynamic? How comprehensive should the model be? Which state variables are important; which have such a negligible effect that they can be ignored? This is an intellectually challenging but rewarding activity that should not be avoided just because it is hard to do.
- (3) Convert the conceptual model into a *specification* model. If this step is done well, the remaining steps are made much easier. If instead this step is done poorly (or not at all) the remaining steps are probably a waste of time. This step typically involves collecting and statistically analyzing data to provide the input models that drive the simulation. In the absence of such data, the input models must be constructed in an ad hoc manner using stochastic models believed to be representative.
- (4) Turn the specification model into a *computational* model, a computer program. At this point, a fundamental choice must be made — to use a general-purpose programming language or a special-purpose simulation language. For some this is a religious issue not subject to rational debate.
- (5) *Verify*. As with all computer programs, the computational model should be consistent with the specification model — did we implement the computational model correctly? This verification step is not the same as the next step.
- (6) *Validate*. Is the computational model consistent with the system being analyzed — did we build the right model? Because the purpose of simulation is insight, some (including the authors) would argue that the *act* of developing the discrete-event simulation model — steps (2), (3), and (4) — is frequently as important as the tangible *product*. However, given the blind faith many people place in any computer generated output the validity of a discrete-event simulation model is always fundamentally important. One popular non-statistical, Turing-like technique for model validation is to place actual system output alongside similarly formatted output from the computational model. This output is then examined by an expert familiar with the system. Model validation is indicated if the expert is not able to determine which is the model output and which is the real thing. Interactive computer graphics (animation) can be very valuable during the verification and validation steps.

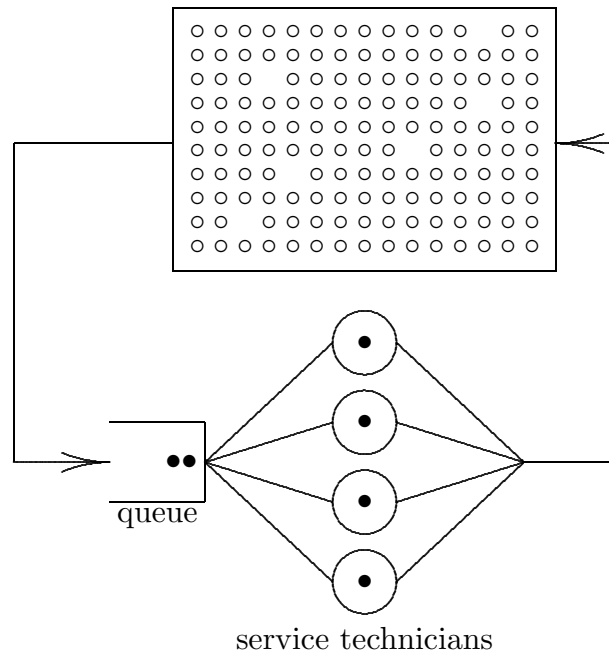
Example 1.1.1 The following *machine shop* model helps illustrate the six steps in Algorithm 1.1.1. A new machine shop has 150 identical machines; each operates continuously, 8 hours per day, 250 days per year until failure. Each machine operates independently of all the others. As machines fail they are repaired, in the order in which they fail, by a service technician. As soon as a failed machine is repaired, it is put back into operation. Each machine produces a net income of \$20 per hour of operation. All service technicians are hired at once, for 2 years, at the beginning of the 2-year period with an annual salary expense of \$52,000. Because of vacations, each service technician only works 230 8-hour days per year. By agreement, vacations are coordinated to maximize the number of service technicians on duty each day. How many service technicians should be hired?

- (1) The objective seems clear — to find the number of service technicians for which the profit is maximized. One extreme solution is to hire one technician for each machine; this produces a huge service technician overhead but maximizes income by minimizing the amount of machine down-time. The other extreme solution is to hire just one technician; this minimizes overhead at the potential expense of large down-times and associated loss of income. In this case, neither extreme is close to optimal for typical failure and repair times.
- (2) A reasonable conceptual model for this system can be expressed in terms of the state of each machine (failed or operational) and each service technician (busy or idle). These state variables provide a high-level description of the system at any time.
- (3) To develop a specification model, more information is needed. Machine failures are random events; what is known (or can be assumed) about the time between failures for these machines? The time to repair a machine is also random; what, for example, is the distribution of the repair time? In addition, to develop the associated specification model some systematic method must be devised to simulate the time evolution of the system state variables.
- (4) The computational model will likely include a simulation clock data structure to keep track of the current simulation time, a queue of failed machines and a queue of available service technicians. Also, to characterize the performance of the system, there will be statistics gathering data structures and associated procedures. The primary statistic of interest here is the total profit associated with the machine shop.
- (5) The computational model must be verified, usually by extensive testing. Verification is a software engineering activity made easier if the model is developed in a contemporary programming environment.
- (6) The validation step is used to see if the verified computational model is a reasonable approximation of the machine shop. If the machine shop is already operational, the basis for comparison is clear. If, however, the machine shop is not yet operational, validation is based primarily on consistency checks. If the number of technicians is increased, does the time-averaged number of failed machines go down; if the average service time is increased, does the time-averaged number of failed machines go up?

System Diagrams

Particularly at the conceptual level, the process of model development can be facilitated by drawing system diagrams. Indeed, when asked to explain a system, our experience is that, instinctively, many people begin by drawing a system diagram. For example, consider this system diagram of the machine shop model in Example 1.1.1.

Figure 1.1.2.
Machine shop
system diagram.



The box at the top of Figure 1.1.2 represents the pool of machines. The composite object at the bottom of the figure represents the four service technicians and an associated single queue. Operational machines are denoted with a \circ and broken machines with a \bullet . Conceptually, as machines break they change their state from operational (\circ) to broken (\bullet) and move along the arc on the left from the box at the top of the figure to the queue at the bottom of the figure. From the queue, a broken machine begins to be repaired as a service technician becomes available. As each broken machine is repaired, its state is changed to operational and the machine moves along the arc on the right, back to the pool of operational machines.*

As time evolves, there is a continual counter-clockwise circulation of machines from the pool at the top of Figure 1.1.2 to the service technicians at the bottom of the figure, and then back again. At the “snapshot” instant illustrated, there are six broken machines; four of these are being repaired and the other two are waiting in the queue for a service technician to become available.

* The movement of the machines to the servers is conceptual, as is the queue. In practice, the servers would move to the machines and there would not be a physical queue of broken machines.

In general, the application of Algorithm 1.1.1 should be guided by the following observations.

- Throughout the development process, the operative principle should always be to make every discrete-event simulation model as simple as possible, but never simpler. The goal is to capture only the relevant characteristics of the system. The dual temptations of (1) ignoring relevant characteristics or (2) including characteristics that are extraneous to the goals of the model, should be avoided.
- The actual development of a complex discrete-event simulation model will not be as sequential as Algorithm 1.1.1 suggests, particularly if the development is a team activity in which case some steps will surely be worked in parallel. The different characteristics of each step should always be kept clearly in mind avoiding, for example, the natural temptation to merge steps (5) and (6).
- There is an unfortunate tendency on the part of many to largely skip over steps (1), (2), and (3), jumping rapidly to step (4). Skipping these first three steps is an approach to discrete-event simulation virtually certain to produce large, inefficient, unstructured computational models that cannot be validated. Discrete-event simulation models should *not* be developed by those who like to think a little and then program a lot.

1.1.3 SIMULATION STUDIES

Algorithm 1.1.2 Following the successful application of Algorithm 1.1.1, use of the resulting computational model (computer program) involves the following steps.

- (7) Design the simulation experiments. This is not as easy as it may seem. If there are a significant number of system parameters, each with several possible values of interest, then the combinatoric possibilities to be studied make this step a real challenge.
- (8) Make production runs. The runs should be made systematically, with the value of all initial conditions and input parameters recorded along with the corresponding statistical output.
- (9) Analyze the simulation results. The analysis of the simulation output is statistical in nature because discrete-event simulation models have stochastic (random) components. The most common statistical analysis tools (means, standard deviations, percentiles, histograms, correlations, etc.) will be developed in later chapters.
- (10) Make decisions. Hopefully the results of step (9) will lead to decisions that result in actions taken. If so, the extent to which the computational model correctly predicted the outcome of these actions is always of great interest, particularly if the model is to be further refined in the future.
- (11) Document the results. If you really did gain insight, summarize it in terms of specific observations and conjectures. If not, why did you fail? Good documentation facilitates the development (or avoidance) of subsequent similar system models.

Example 1.1.2 As a continuation of Example 1.1.1, consider the application of Algorithm 1.1.2 to a verified and validated machine shop model.

- (7) Since the objective of the model is to determine the optimal number of service technicians to hire to maximize profit, the number of technicians is the primary system parameter to be varied from one simulation run to the next. Other issues also contribute to the design of the simulation experiments. What are the initial conditions for the model (e.g., are all machines initially operational)? For a fixed number of service technicians, how many replications are required to reduce the natural sampling variability in the output statistics to an acceptable level?
- (8) If many production runs are made, management of the output results becomes an issue. A discrete-event simulation study can produce *a lot* of output files which consume large amounts of disk space if not properly managed. Avoid the temptation to archive “raw data” (e.g., a detailed time history of simulated machine failures). If this kind of data is needed in the future, it can always be reproduced. Indeed, the ability to reproduce previous results *exactly* is an important feature which distinguishes discrete-event simulation from other, more traditional, experimental sciences.
- (9) The statistical analysis of simulation output often is more difficult than classical statistical analysis, where observations are assumed to be *independent*. In particular, time-sequenced simulation-generated observations are often correlated with one another, making the analysis of such data a challenge. If the current number of failed machines is observed each hour, for example, consecutive observations will be found to be significantly positively correlated. A statistical analysis of these observations based on the (false) assumption of independence may produce erroneous conclusions.
- (10) For this example, a graphical display of profit versus the number of service technicians yields both the optimal number of technicians and a measure of how sensitive the profit is to variations about this optimal number. In this way a policy decision can be made. Provided this decision does not violate any external constraints, such as labor union rules, the policy should be implemented.
- (11) Documentation of the machine shop model would include a system diagram, explanations of assumptions made about machine failure rates and service repair rates, a description of the specification model, software for the computational model, tables and figures of output, and a description of the output analysis.

Insight

An important benefit of developing and using a discrete-event simulation model is that valuable insight is acquired. As conceptual models are formulated, computational models developed and output data analyzed, subtle system features and component interactions may be discovered that would not have been noticed otherwise. The systematic application of Algorithms 1.1.1 and 1.1.2 can result in better actions taken due to insight gained by an increased understanding of how the system operates.

1.1.4 PROGRAMMING LANGUAGES

There is a continuing debate in discrete-event simulation — to use a general-purpose programming language or a (special-purpose) simulation programming language. For example, two standard discrete-event simulation textbooks provide the following contradictory advice. Bratley, Fox, and Schrage (1987, page 219) state “. . . for any important large-scale real application we would write the programs in a standard general-purpose language, and avoid all the simulation languages we know.” In contrast, Law and Kelton (2000, page 204) state “. . . we believe, in general, that a modeler would be prudent to give serious consideration to the use of a simulation package.”

General-purpose languages are more flexible and familiar; simulation languages allow modelers to build computational models quickly. There is no easy way to resolve this debate in general. However, for the specific purpose of this book — learning the principles and techniques of discrete-event simulation — the debate is easier to resolve. Learning discrete-event simulation methodology is facilitated by using a familiar, general-purpose programming language, a philosophy that has dictated the style and content of this book.

General-Purpose Languages

Because discrete-event simulation is a specific instance of scientific computing, any *general-purpose* programming language suitable for scientific computing is similarly suitable for discrete-event simulation. Therefore, a history of the use of general-purpose programming languages in discrete-event simulation is really a history of general-purpose programming languages in scientific computing. Although this history is extensive, we will try to summarize it in a few paragraphs.

For many years FORTRAN was the primary general-purpose programming language used in discrete-event simulation. In retrospect, this was natural and appropriate because there was no well-accepted alternative. By the early 80’s things began to change dramatically. Several general-purpose programming languages created in the 70’s, primarily C and Pascal, were as good as or superior to FORTRAN in most respects and they began to gain acceptance in many applications, including discrete-event simulation, where FORTRAN was once dominant. Because of its structure and relative simplicity, Pascal became the de facto first programming language in many computer science departments; because of its flexibility and power, the use of C became common among professional programmers.

Personal computers became popular in the early 80’s, followed soon thereafter by increasingly more powerful workstations. Concurrent with this development, it became clear that networked workstations or, to a lesser extent, stand-alone personal computers, were ideal discrete-event simulation engines. The popularity of workstation networks then helped to guarantee that C would become the general-purpose language of choice for discrete-event simulation. That is, the usual workstation network was Unix-based, an environment in which C was the natural general-purpose programming language of choice. The use of C in discrete-event simulation became wide-spread by the early 90’s when C became standardized and C++, an object-oriented extension of C, gained popularity.

In addition to C, C++, FORTRAN, and Pascal, other general-purpose programming languages are occasionally used in discrete-event simulation. Of these, Ada, Java, and (modern, compiled) BASIC are probably the most common. This diversity is not surprising because every general-purpose programming language has its advocates, some quite vocal, and no matter what the language there is likely to be an advocate to argue that it is ideal for discrete-event simulation. We leave that debate for another forum, however, confident that our use of ANSI C in this book is appropriate.

Simulation Languages

Simulation languages have built-in features that provide many of the tools needed to write a discrete-event simulation program. Because of this, simulation languages support rapid prototyping and have the potential to decrease programming time significantly. Moreover, animation is a particularly important feature now built into most of these simulation languages. This is important because animation can increase the acceptance of discrete-event simulation as a legitimate problem-solving technique. By using animation, dynamic graphical images can be created that enhance verification, validation, and the development of insight. The most popular discrete-event simulation languages historically are GPSS, SIMAN, SLAM II, and SIMSCRIPT II.5. Because of our emphasis in the book on the use of general-purpose languages, any additional discussion of simulation languages is deferred to Appendix A.

Because it is not discussed in Appendix A, for historical reasons it is appropriate here to mention the simulation language Simula. This language was developed in the 60's as an object-oriented ALGOL extension. Despite its object orientation and several other novel (for the time) features, it never achieved much popularity, except in Europe. Still, like other premature-but-good ideas, the impact of Simula has proven to be profound, including serving as the inspiration for the creation of C++.

1.1.5 ORGANIZATION AND TERMINOLOGY

We conclude this first section with some brief comments about the organization of the book and the sometimes ambiguous use of the words *simulation*, *simulate*, and *model*.

Organization

The material in this book could have been organized in several ways. Perhaps the most natural sequence would be to follow, in order, the steps in Algorithms 1.1.1 and 1.1.2, devoting a chapter to each step. However, that sequence is not followed. Instead, the material is organized in a manner consistent with the experimental nature of discrete-event simulation. That is, we begin to model, simulate, and analyze simple-but-representative systems as soon as possible (indeed, in the next section). Whenever possible, new concepts are first introduced in an informal way that encourages experimental self-discovery, with a more formal treatment of the concepts deferred to later chapters. This organization has proven to be successful in the classroom.

Terminology

The words “model” and “simulation” or “simulate” are commonly used interchangeably in the discrete-event simulation literature, both as a noun and as a verb. For pedagogical reasons this word interchangeability is unfortunate because, as indicated previously, a “model” (the noun) exists at three levels of abstraction: conceptual, specification, and computational. At the computational level, a system model is a computer program; this computer program is what most people mean when they talk about a system simulation. In this context a simulation and a computational system model are equivalent. It is uncommon, however, to use the noun “simulation” as a synonym for the system model at either the conceptual or specification level. Similarly, “to model” (the verb) implies activity at three levels, but “to simulate” is usually a computational activity only.

When appropriate we will try to be careful with these words, generally using simulation or simulate in reference to a computational activity only. This is consistent with common usage of the word *simulation* to characterize not only the computational model (computer program) but also the computational process of using the discrete-event simulation model to generate output statistical data and thereby analyze system performance. In those cases when there is no real need to be fussy about terminology, we will yield to tradition and use the word simulation or simulate even though the word model may be more appropriate.

1.1.6 EXERCISES

Exercise 1.1.1 There are six leaf nodes in the system model tree in Figure 1.1.1. For each leaf node, describe a specific example of a corresponding physical system.

Exercise 1.1.2 The distinction between model *verification* and model *validation* is not always clear in practice. Generally, in the sense of Algorithm 1.1.1, the ultimate objective is a valid discrete-event simulation model. If you were told that “this discrete-event simulation model had been verified but it is not known if the model is valid” how would you interpret that statement?

Exercise 1.1.3 The *state* of a system is important, but difficult to define in a general context. (a) Locate at least five contemporary textbooks that discuss system modeling and, for each, research and comment on the extent to which the technical term “state” is defined. If possible, avoid example-based definitions or definitions based on a specific system. (b) How would you define the state of a system?

Exercise 1.1.4 (a) Use an Internet search engine to identify at least 10 different simulation languages that support discrete-event simulation. (Note that the ‘-’ in discrete-event is not a universal convention.) Provide a URL, phone number, or mailing address for each and, if it is a commercial product, a price. (b) If you tried multiple search engines, which produced the most meaningful hits?