UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO

COMPUTER SCIENCE DEPARTMENT
COMPUTER SCIENCE DEPARTMENT
COMPUTER SCIENCE DEPARTMENT

A First Course in Simulation

G. R. Sager
J. W. Wong

January 1978
CS-78-03

A First Course in Simulation

G. R. Sager
J. W. Wong

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

General Comments on the Evaluation of Simulation Assignments and Projects

The following comments are intended to serve as a set of guidelines for the student to use in the preparation of assignments. The outline may not be appropriate in all cases, but it will indicate the general content expected of all assignments.

TOOLS: First and foremost, discover and use the tools available. In particular, a text editor such as QED or the CMS editor will be useful in preparation of program source, data files, and the report text. A simulation language such as GPSS or SIMSCRIPT is helpful, but in their absence one may use FORTRAN-based subroutine packages (GASP, for example) or one may find it worthwhile to invest time in developing a few general tools before actually embarking on the project. For the report itself, a document preparation tool (ROFF or SCRIPT) allows one to prepare and easily revise a report. The resulting report will have quite a professional appearance. Statistics packages can aid one in the evaluation of data, and plotting packages can supplement a report with neatly prepared summaries of the data and results. (In fact, the ST package on the H66/60 is quite well suited to this application for the purposes of this course, and is extremely easy to use.)

GENERAL OUTLINE OF A REPORT

Abstract: Start the report with a short (50 - 150 words) subjective resume of what the simulation study has to say to the reader. That is, he should be given an idea of what to look for while reading the report.

I. External Documentation

A. *Describe the system* being studied. If necessary, educate the reader in the terminology and other important aspects of the system.

B. *State the objectives* of the study. These may usually be classified into one or more of the following:

1. Prediction of system performance for a given set of inputs.

2. Evaluation of alternate system design strategies.

3. Control of input for a given level of performance.

C. *Describe the model* of the system.

1. *Abstract* into environment, subsystems, entities, attributes, events and activities. Give tables of entities and their attributes. Make block diagrams to illustrate the interrelationships of the subsystems, entities, events and activities. If the abstraction makes any simplifying assumptions (i.e. it may leave out some things which are normally present in the real system), bring them to the reader's attention and justify the omission.

2. *Describe* the model in words. the description should reflect the structure of the model as a logical and coherent organization of fundamentals - i.e. the model is built as a structure of interacting subsystems using either a top down or bottom up approach.

3. *Validate* the model. This may not be feasible in all cases. When possible, try to make direct comparisons between the model and the real system. This may involve the use of statistical methods to evaluate the acceptability of the model's results.

D. *Interpret the results* or *Illustrate use of the model*

1. *Present* the results in the form of graphs, Kiviat graphs, histograms, tables, etc. Note that selection of a good graphical technique to emphasize the results can be difficult; try several, but use only the best.
Important: label axes and give units!

2. *Explain* the outcome: give arguments to show that the outcome is reasonable. If exact analyses can be applied, use them. Perhaps exact results of similar systems can be used as approximations. Demonstrate that results lie within any derivable upper or lower bounds. As a last resort, make arguments based on intuition.

3. *Evaluate* the results: it isn't enough to say that something is "better" or "different"; there must be some confidence estimate attached to the statement. This involves the experimental technique of formulating hypotheses and accepting or rejecting them on the basis of statistical formulae applied to the results.

4. *Document* the measures and statistics: if a measure or statistic appears in the study, it *must* be used and it *must* be discussed. Do not overpower the reader with charts, graphs and tables full of useless numbers. If the measure or statistic is unfamiliar to the reader or if special pains are required to gather the data, be sure to include an explanation of the technique.

5. *Warn* the reader against potential misuses of the model. In particular, if certain ranges of the input may cause inaccurate or less accurate results, give adequate cautions.

II. Internal Documentation

Note that the computer program is not the "model" in a simulation study; it is a realization of the model. Thus, a customer interested in the system and the model may care little for the "hacking" required to realize it as a program. Moral: document the program separately.

A. *Program listing*
Use paragraphing to emphasize the structured flow of control. Use subroutines not only to reduce the coding but also to isolate logical components. Comments should be inserted to aid clarity to the flow of control. Declaration and typing information for variables should be grouped for quick access to all necessary information. Use mnemonic variable names.

B. *Supplementary to the listing*

1. *Data Structures:* give all typing information and a verbal description of the uses of all variables and data structures. Indicate which subroutines change the data and which make decisions based on the data.

2. *Subroutines:* describe what each does and what its formal parameters are. Indicate which formal parameters and/or common variables are inputs, which are outputs and which are both. List all local variables and their uses.

3. *Key* all variables and subroutines to the model as described in section I.C.

C. *Certify the program*
Since the program is a realization of the model, we must assure ourselves that the realization is free of bugs. There are a great number of ways of increasing our confidence in a program. In addition to the ones which apply to standard programs, the following techniques should prove useful:

1. *Run the program* with known inputs to produce easily predicted outputs. For example, some models have analytical solutions under certain conditions; verify that the program will yield those results. Note that this may require inputs which would make no sense in terms of the real system (for example, unrealistic distributions for interarrival time and service time).

2. *Perform redundant calculations.* Especially in the case of many performance measures there will often be two or more fairly independent ways of obtaining the same result. For example, the "utilization factor" can be computed as the total busy time divided by the elapsed time as well as by mean service time divided by mean interarrival time. Verify that such redundancies agree.

3. *Validation of the model* of course says something about the program, but don't use this as the first and/or only attempt to certify the program; validation by itself is difficult enough without trying to debug at the same time.

III. Bibliography

You should have done some looking around in various journals or perhaps you asked questions of persons involved in the system you are studying; if so, you should reference these articles or "private communications" in a bibliography. Failure to do this is considered to be the most heinous of sins in academia.

CHAPTER 1
Systems, Models, and Simulation by Computer

1.1 Introduction

Many situations we encounter in everyday life are readily characterized as a *queueing system:* namely, there are *customers* who require a service which can be provided only by a *server.* Sometimes the queueing situation is so bad that we get frustrated and disatisfied with the manner in which the system is operated. For example, in a busy supermarket, we may consider ways in which the service might be improved such that customers do not spend as much time waiting in line for the cashier. As customer, we may employ selfish strategies: in a supermarket with separate queues in front of each cashier, if we notice that another queue is shorter than our own, we reduce our wait time by switching to it. Also, when we find that the queues are too long, we may leave without service. Perhaps we can argue that it is more fair that customers with only a few items should be allowed to use an express checkout. If the management provided more cashiers, the length of the queue (or queues) would be shorter, thereby decreasing every customer's waiting time.

On the other hand, as the manager of the supermarket, we would realize that the addition of cashiers increases the cost of operation. We are confronted with conflicting problems: poor service may lose customers, but the price of improved service may cause customers to take their business elsewhere.

In either role, we may feel that we either know or could discover a better way of doing things. The problem is: how do we convince ourselves (and others) that we are right? Undoubtedly some experimentation is required, but we may not be in a position to experiment with a queueing system on a large enough scale to try out our ideas. Given that we can conduct experiments, we must be relatively certain that the experiments are true reflections of what will happen when the system is operating in a less controlled, real-life environment. Moreover, to simply say that our experimental system is "different" or "better" is not enough; we must determine that the difference is in some sense significant. These problems are quite similar to those of researchers in the experimental sciences, who attempt to develop theories which are better explanations of real-world phenomena than existing theories. When the constraint is added that one cannot perform experiments with the system itself, then a technique known as *simulation* must be used. With simulation, the researcher performs his experiments on a model of the system rather than on the system itself. Although this avoids problems which may have been insurmountable with the system, it makes the task of convincing more difficult, for now the question arises whether or not the experiments on the model are equivalent to similar experiments on the system.

In the remaining sections of this chapter, we will introduce some terminology used in simulation, and outline the basic steps that one would typically go through in a simulation study.

1.2 Systems

Informally, a *system* is a collection of interrelated *entities*; and each entity is characterized by a set of possibly interrelated *attributes*. We thus define an entity to be a component of the system; it may in fact be a system if considered by itself, but interests us only as a subsystem of the system we are studying. An attribute is then a property of the system or of an entity in the system. As an example, consider a single checkout counter in a supermarket. We might identify the following entities and their attributes:

entities	*attributes*
checkout counter (system)	arrival rate of customers distribution of items purchased length of queue
cashier	income rate of work
customer	number of items purchased

Attributes may be expressed as constants ("the cashier's income is $10,000") or as distributions ("10% of arriving customers purchase 1 item, 20% purchase 2 items, 15% purchase 3 items, etc.").

In a queueing system, the *arrival rate* is commonly denoted by the symbol λ; the units of this attribute are customer arrivals per time unit. The inverse of arrival rate is *interarrival time*, measured as time units per customer arrival (i.e., the elapsed time time between arrivals) and denoted by $1/\lambda$. The rate of work, or *service time* is denoted by the symbol μ and is measured as customer services per time unit. The inverse of service rate is *service time*, measured as time units per customer service and denoted by $1/\mu$.

In this system, we suspect that cashier's rate of doing work may be related to his or her income. The interrelationship of the cashier and customer entities is straightforward: the amount of time a customer spends with the cashier is directly related to the rate of work of the cashier and the number of items purchased by the customer. The rule which determines the order of customers in the queue is known as the *discipline* of the queue. The queueing discipline, along with other information such as the rate of work of the cashier, the number of customers in the queue and the number of items purchased by each customer determine the waiting time customers experience.

If we consider that this system is actually one of many "checkout subsystems" in a larger supermarket system, then we note that the subsystems are subject to interrelationships; in particular, the queue lengths of the subsystems tend to be equal because customers will usually join the shortest queue and will perhaps move from one queue to another as the lengths change.

The *state* of a system is an instantaneous description of the system based on its entities and attributes. At any point in time, the state of the system is the set of information which would be sufficient to "restart" the operation of the system without ill effects; that is, by simply setting the system to that state, it should continue to

operate as though it had arrived at that state by natural means. In our supermarket checkout counter example, we can define the state to be the length of the queue and the number of items purchased by each customer in the queue. The values of the other attributes do not change during the operation of the system and therefore do not constitute state information. These attributes are known as *system parameters* and determine how the system will respond to a given state.

We have perhaps been overzealous in identifying the entities and attributes of the checkout counter; the income attribute is quite well characterized by the rate of doing work alone unless we intend to study the trade-off between using more skilled, higher paid cashiers versus less skilled, lower paid ones. On the other hand, if our intentions are for a much more detailed study, the entities and attributes suggested above may not be adequate. Thus, the identification of entities and attributes requires a bit of forethought about the goal of the study.

A system may be considered to be discrete or continuous, depending upon the way it changes from one state to another. In a *discrete system*, the state changes by discrete values and at discrete instants. Our example of a supermarket checkout counter is discrete: for example, the number of customers in the queue increases by 1 when a customer joins the queue, and decreases by 1 when a customer finishes service. In a *continuous system*, the state changes continuously and smoothly. Some examples of such systems are:

1. A dam: state is water level.
2. The suspension of an automobile wheel: state is distance from rest position and velocity (up or down).
3. An airplane in flight: state is altitude, attitude and velocity (in 3 dimensions).

The state information suggested above is minimal: it is likely that in any detailed study, the states would be far more complex.

In this text, we will restrict ourselves to systems whose state changes are best described as discrete.

1.3 Models

The first step in studying a system is to build a model. Informally, a *model* is an abstraction of the system under study. This abstraction is obtained by capturing the essential characteristics of the system, i.e., characteristics that would affect system performance the most (obviously this requires some judgment).

There are different types of models. A *physical model* is a scaled down replica of the system, e.g., a model airplane. A *mathematical model* is a characterization of a system by a set of well-defined relationships (i.e. equations, algorithms and/or operations on data structures). If we can deduce from this set of equations an explicit solution to the performance measures of interest, we have an *analytic model.* An example for this is Newton's laws of motion. If it is not possible to get an explicit solution but numerical methods can be used, then we have a *numerical model.* Numerical models are typically used when a continuous system is modelled by a system of differential equations or when a discrete system is modelled by interconnected queues which are difficult or impossible to solve analytically.

Models can be further classified as to whether they are deterministic or stochastic. In a *deterministic model,* any valid input will have a precisely determined effect on the state of the model (as in the logic circuits of a computer). In a *stochastic model,* any valid input will result in one of many possible effects on the state of the model, due to some randomness which is built into the model. Generally, this randomness is introduced into the model to account for some aspect of the system which either cannot or need not be modelled in a precise deterministic way. For example, were we to attempt to model in every detail the events and activities involved in the ringing up of a single item at our grocery store checkout counter, the range of possibilities quickly becomes unmanageable: perhaps the price is not marked or smudged so a price check is required, perhaps the item is produce and the price must be looked up on a sheet at the register, perhaps the checker must turn the item over several times in order to find the price, etc. It is much simpler and undoubtedly quite effective for most purposes to simply note that the checker requires sometimes more, sometimes less time for an item and account for this as a stochastic element in the model. These stochastic elements typically appear in the abstraction as attributes expressed as distributions.

Another possible breakdown is whether the model is static or dynamic. In a *static model,* state changes are not related to time; for example, the operation of a lever is determined by physical laws which do not involve time. In a *dynamic model,* state changes are related to time; in a model of our checkout counter system, for example, we must account for the time spent waiting in the queue and the time spent with the cashier.

In this text, we will concentrate on mathematical models which are numeric, stochastic and dynamic.

There are many reasons for experimenting with a model instead of the real system. Some important ones are:

1. The system may only be in the planning stage and is therefore not available for study.
2. A model often leads to improved understanding of the system.
3. Once a model is developed, we may get results faster. Experimenting with the real system may take months or years to get the desired results.
4. A model enables us to identify the problem areas before any real change is made to the system, this is especially useful when the change involves a substantial amount of capital investment.
5. A model is easier to manipulate than the real system; we can efficiently try out various design strategies and compare performance.
6. The real system may be too risky to experiment with, e.g., an air-defense weapon, or a nuclear reaction.

The fact that we experiment with a model rather than the system itself leads to another problem: we are assuming that experiments on the model are equivalent to experiments on the system, and must therefore supply evidence to support this assumption. Thus, we must conduct experiments and use statistical methods to show that the model and the system are equivalent to a degree which will justify the further conclusions we draw from the model; this is known as *validation.* Of course, in cases where the system is not available for experimentation, validation is not possible.

1.4 Simulation by computer

Once the model of a system has been developed, the experimenter must obtain result from it for comparison with the original system or with other versions of the model. One method of doing this would be to use a pencil, paper and dice (for the stochastic elements) and run through the steps called for by the model by hand. This technique is quite acceptable for a small study, or for preliminary tests of model validity; it can avoid a great deal of waste spent in writing and debugging a computer program. However, most simulation studies will require a large number of operations and a large number of variables, so the experimenter will find that the accuracy and speed of a digital computer is necessary. There are two dangers to this course of action: first, large computer programs require a large investment of time and effort to implement and, second, the program is not the model. The second point is rather important, for it means that there is yet another potential source of error; if our model will not validate, for example, is it because the model itself is improperly conceived or is it because the program we have written does not implement it correctly (i.e. a bug)? Thus it is important as much as possible to separate the model from the program in order to prevent confusion between validation and debugging.

1.5 Basic steps in a simulation study

We conclude this chapter by giving the steps that one would typically go through in a simulation study. It should be noted that these steps refer to material which will covered in later chapters. It is recommended that this section be frequently reviewed as progress is made through the later chapters.

1. Outline the objectives of the study. This will usually consist of one or more of the following:
   (a) Prediction of system performance for a given set of inputs.
   (b) Evaluation of alternate system design strategies.
   (c) Control of input for a given level of performance.

2. Have a clear understanding of the system under investigation and identify:
   (a) The essential characteristics
   (b) The system parameters
   (c) The measures which best characterize "system performance"

3. Design a model for the system. This includes the abstraction of the system into its environment, subsystems, entities, attributes, sets, states, events, and activities and the characterization of their interrelationships. Any simplifying assumption made should be mentioned and justified. The effect of these assumptions on the accuracy of the model should also be discussed. For a complex system, a structural approach should be used: e.g., a top-down refinement of the system into subsystems and then to entities, attributes, events, etc., or a bottom-up approach of modelling subsystems, and then combining subsystem models into a total system model.

4. Design and implement a simulation program for the model developed.

5. Run acceptance tests on the program. Quite often it will be possible to apply inputs to the program which will result in predictable results, even though these inputs may not be representative of the inputs to the real system. One should also apply the standard program debugging techniques, such as testing individual modules of the program for correctness and being sure to try the limiting conditions on loops and data structures.

6. Estimate the system parameters for the model. A common method is to observe the real system and characterize the system parameters as constants or probability distributions. In case that the system is still in the planning stage, we can estimate the parameters from observations made in existing systems that are similar in nature.

7. Validate the model. It is important to note that the model is only an abstraction of the system. It must be validated before it can be used as a tool for studying system behavior. Validation is the process of checking the consistency of the model with the real system under the existing condition. If a model cannot reproduce the system behavior as is, one can hardly expect it to give meaningful results under new conditions. During the validation process, one might discover that the model does not represent correctly a certain aspect of the system. This often provides information for the analyst to refine his model. The development of a useful model is therefore a repeated process of model refinement, and it is not likely that one can get a good model after the first pass.

8. Design the experiments. It is not sufficient to run the model once; since random numbers are used in the execution, one may rerun the program with a new set of random numbers and expect to see results which differ from other runs. If the results differ greatly from one run to the next, they may reflect similar "erratic" behavior in the real system or perhaps indicate that the experiment is not being conducted properly. In either case, one must be prepared to redesign the model, program and/or the experimental technique as the situation indicates.

9. Run the simulation program (which is an implementation of the model) with selected values for the system parameters, and collect data for the desired performance measures. Present the output data in the form of graphs, Kiviat graphs, histograms, tables, etc.

10. Analyse and interpret the output data. The graphs and tables which present the results should be discussed, and statistical methods used to support the conclusions of the study.

These steps should not be persued strictly in sequence; it is possible to do several steps in parallel. For example, one may begin collecting data (step 6) while still observing the system to gain understanding (step 2). However, one should always be ready to discard false starts and start over. Even the most experienced analyst will make mistakes, and if he has any advantage over the beginner it is undoubtedly a willingness to recognize his errors and correct them rather than stubbornly attacking the problem from the first approach taken. For the beginner, a bottom-up design is usually best; this is perhaps because it allows one to conduct "mini studies" on the subsystems as they are developed, thereby gaining some experience before the entire study is undertaken.

CHAPTER 2
Discrete Event Simulation

In this chapter, we introduce the terminology used in discrete event simulation by describing in detail the basic structure of a simple discrete event simulation program. We will base our discussion on the example system of a supermarket checkout counter introduced in Chapter 1.

2.1 Terminology

We define the *environment* of the system to be the important entities, the important attributes of those entities, and the range of values that these attributes can take on. By the term "important" we mean that the entity or attribute was not eliminated in the process of abstraction. An analyst often fixes the values of some attributes and conducts experiments to determine the effect of varying other attributes. These other attributes are called *input parameters*. In queueing systems, two typical selections for input parameters are *interarrival time or arrival rate* (denoted by $1/\lambda$ or λ, respectively) and *service time or service rate* (denoted by $1/\mu$ or μ, respectively). In our example system, the interarrival time is the time which elapses from when one customer joins the queue until the next customer joins (i.e. the attribute arrival rate of customers); the service time is the total time a customer spends with the cashier, this time is a function of the rate of work of the cashier and the number of items purchased by the customer being served.

Consider the modelling of a single checkout counter in a supermarket. A table of entities and attributes for this system can be found in section 1.2. In designing our model, we remove the cashier's income to simplify the model. We thus have the following table for entities and their attributes:

entities	*attributes*
checkout counter (system)	arrival rate of customers distribution of items purchased length of queue
cashier	rate of work
customer	number of items purchased

Set is a term used to denote a queue. Each set has an *owner* and contains *members*. The owner and members must be entities. For our example of a checkout counter, there is only one set:

set	*owner*	*members*
queue	cashier	customers

In discrete simulation, an *event* is an instantaneous operation which changes the state of the system; an *activity* is started and terminated by related events. Note that by definition, an event takes no time, therefore it is theoretically impossible to stop the system during an event; to do so might result in a state which would be considered illegal. A useful heuristic for identifying the events and activities of a system is to imagine a trip through the system from the viewpoint of a customer entity. For our checkout counter example, the manner in which customers move through the system can be characterized by the following sequence of events and activities:

events	*activities*
arrive at counter	
	wait in queue
start service at cashier	
	receive service from cashier
depart from system	

Note that we have interleaved the table entries such that every activity is bracketed by events; this is done to emphasize the fact that activities begin and end with events. This method of presentation is effective when the sequences of events and activities occur serially and in a well-determined order. Later, we will encounter systems in which events may start and/or end more than one activity and events may "decide" which of several activities to begin, based on the state of the system at the time of the event. It is important to realize that this presentation is from a single customer's viewpoint, and that many customers will be at various stages of the activities, as indicated in figure 2.1.a.

Even in the highly simplified system we are considering, the table of events and activities hides two phenomena which will become apparent when we later code the program for the simulation model:

1. If the system is empty when the customer arrives, the 'wait in queue' activity takes zero time and the events 'arrive at counter' and 'begin service at cashier' occur at the same time.
2. If the cashier is busy when a customer arrives at the counter, that customer's 'begin service at cashier' event occurs at the same time as the event 'depart from system' for the customer preceding him according to the queueing discipline.

2.2 Event scheduling approach

In writing a program to implement the model for a discrete event simulation model, we take advantage of the fact that the state of the system remains unchanged between events (or during activities), so it is only necessary to consider times at which events occur. A clock is used to keep track of *simulated time*, which corresponds to *real time* in the system being modelled. When we speak of time in discussions which follow, we mean simulated time unless otherwise specified. The basic technique is to advance the clock to the time of next event and modify the state as indicated by the event.

There are several "world views" in use by simulation packages available today. One is the *event scheduling* approach used by packages such as SIMSCRIPT. A simple explanation of this approach is that the underlying structure provided by the package attempts to invoke the *event routines* at the appropriate simulated times. Thus the user writes his simulation program as a group of subroutines which define how state changes occur at each event. An alternative world view is called *process interaction* and is typified by the GPSS package. In the process interaction approach, the underlying structure provided by the package attempts to move the "transactions" (or entities) to the completion of their activities. The user then writes his simulation program as a list of statements which describe the activities of the system.

For this chapter, emphasis will be given to the event scheduling approach of discrete simulation. In order to better understand some of the underlying structure provided by packages using this approach, we consider a FORTRAN program designed to implement the checkout counter system developed in previous sections and summarized in section 2.1. The program listing appears at the end of this section.

The main program of the accompanying FORTRAN program is responsible for setting the variable CLOCK to the current simulated time and calling the event routines. Its operation is as follows:

1. Initialize (to be discussed later).
2. Select the next event (indicated by CUREV) from the *event list*; all events scheduled to occur at some future time appear in this list, ordered by the time at which they are to occur.
3. Advance CLOCK to the time of this event (indicated by EVTIME) and delete the event from the event list.
4. According to the event type (indicated by EVTYPE), call the event routine to perform the state modification. Here, we have chosen type 1 to specify the 'arrive at counter' event and type 2 to specify the 'depart from system' event. These event routines may add events to the event list.
5. Go to 2 (statement label 10).

The events outlined in Section 2.1 are implemented as subroutines which change the state of the model by changing the values of variables and/or data structures. We first define how the model changes state when a customer arrives:

Subroutine for 'arrive at counter' event (ARRIVE):

1. Increment count of number of customers in the system (NSYS).
2. Schedule an 'arrive at counter' event to occur at a future time.
3. Determine the number of items purchased by the arriving customer (ITEMS).
4. Enter the customer into the queue, remembering the number of items purchased.
5. If the cashier is not busy (IDLE) call the event routine for 'start service at cashier' (SERVE).

Note that at step 2, the arrival event schedules the next arrival event; this practice is known as *bootstrapping*. It is a very natural way to schedule arrivals when we use the customer arrival rate or interarrival time as an input parameter. Given a distribution for the interarrival time, a random number generator can be used to determine the time of the next arrival event by first generating an interarrival time and then adding it to the current time (which is in CLOCK). For the present, we will use the uniform distribution which has the property that all possible values in an interval (a,b) occur with equal probability. The mean of this distribution is (a+b)/2. In our example program, the desired mean interarrival time is stored in the variable IATIME. We use the subroutine URANDM to obtain random numbers which are uniformly distributed between 0 and 1, and then multiply these numbers by 2*IATIME to get values uniformly distributed between 0 and 2*IATIME (with mean value IATIME). With this distribution "built in" to the program, we need only specify the mean interarrival time as an input.

The alternative to bootstrapping would be to schedule all of the arrival events at the beginning of the simulation (in the subroutine INIT, called by the main program) or to read the next arrival time at each arrival event. The first method is undesirable because it requires a great deal of space to store the events in the event list. In addition, we would either have to collect very detailed records from the system in order to schedule the arrivals just as they occurred in the system, or we would generate the interarrival times randomly in a manner similar to that used for bootstrapping. The second method requires the preparation of a large amount of detailed data about times of arrival events collected from the system. If this data is available, it can be very valuable for a *trace-driven simulation*. One of the main advantages of trace-driven simulation is that it provides assurance that the input to the model is precisely what happened in the system; this increases confidence in the results and improves the chances for validating the model. However, we will see later that when a human is observing a system directly, such detailed information may be difficult to obtain; the main use of trace-driven simulations has been in simulation of computer systems, where detailed accounting records provide the necessary inputs to the model already in computer readable form.

It is not apparent from our program what the unit of measure for time is. This is in fact arbitrary; judging from the nature of the system, it seems reasonable to measure time in minutes. Some simulation packages, such as SIMSCRIPT, allow the user to specify the time units and provide conversions to seconds, minutes, hours, days, weeks, etc. Other packages, such as GPSS, use a single unidentified "time unit" and require the user to do his or her own conversions. We will take the latter approach, though it is easy to imagine how subroutines could be written to do the conversions if we desired such luxuries.

At step 2 of the arrival event, we have arbitrarily chosen to decide the number of items purchased by the arriving customer as uniformly distributed between 1 and 9. As with the interarrival time, this should be considered to be a temporary measure; after we have studied the program sufficiently and convinced ourselves that we understand it and that it works, we can replace these arbitrary distributions with ones observed in the real system.

When a customer has finished receiving service from the cashier, he departs from the system. The state change for this event is defined by the following subroutine:

Subroutine for 'depart from system' event (DEPART):

1. Decrement number of customers in system (NSYS).
2. Mark cashier idle.
3. If there is one or more customers in the queue, call the 'start service at cashier' routine (SERVE) to get one started.

Note that in both the arrival and departure events, we have called the subroutine which corresponds to a customer starting service at the cashier (SERVE). As noted in section 2.1, under certain conditions, this event occurs simulataneously with the 'arrive at counter' or 'depart from system' events. In fact, since the cashier will always provide service when there are customers, the 'start service at cashier' event is never explicitly scheduled (i.e., using SCHED). We refer to this as an *implicit event* (or *conditional event*).

Subroutine for 'start service at cashier' event (SERVE):

1. Mark cashier busy (not IDLE).
2. Remove a customer from the queue, recalling the number of items purchased by that customer.
3. Schedule a 'depart from system' event to mark the termination of the service activity about to begin.

In a typical simulation, one would provide, as input, a distribution for the service time; but we have fixed this in the program to be uniformly distributed between 3 and number of items purchased + 3.

For completeness, we include here a description of the routine SCHED, which makes entries into the event list. In order to keep the event list in a form from which the main program can easily determine the next event, SCHED does what is best described as the insertion step of an insertion sort; each record consists of a key (time at which the event is to occur) and a datum (the type of event). As SCHED looks up the relative position to place the new event record in, it also moves along the slot into which it will place the event record. The algorithm used also insures that in case of ties, the event which was scheduled first in real time will occur first in real time (although the tied events will occur at the same simulated time). This is a fairly primitive approach, as it would be fairly inefficient for large numbers of events whose time of occurence would require the insertion sort to look far into the event list. It also precludes the possiblility of passing arguments to the event routines. Simulation systems such as SIMSCRIPT and GPSS often use more sophisticated techniques to insure efficient event scheduling and to allow the passing of arguments. They also allow facilities for the user to determine what will happen in the case of ties.

```
C   MAIN PROGRAM - CHECKOUT COUNTER SIMULATION
      REAL CLOCK
      COMMON /TIME/ CLOCK
      INTEGER CUREV, EVTYPE(10)
      REAL EVTIME(10)
      COMMON /EVENTS/ CUREV, EVTIME, EVTYPE
      INTEGER TYPE

      CALL INIT
 10   CLOCK = EVTIME(CUREV)
      TYPE = EVTYPE(CUREV)
      CUREV = CUREV - 1
      IF ( TYPE .EQ. 1 )  CALL ARRIVE
      IF ( TYPE .EQ. 2 )  CALL DEPART
      GO TO 10
      END


      SUBROUTINE SCHED (TYPE, WHEN)
      REAL CLOCK
      COMMON /TIME/ CLOCK
      INTEGER CUREV, EVTYPE(10)
      REAL EVTIME(10)
      COMMON /EVENTS/ CUREV, EVTIME, EVTYPE
      INTEGER CUR, TOP, TYPE
      REAL WHEN

      CUR = CUREV
      CUREV = CUREV + 1
      IF ( CUREV .GT. 10 ) GO TO 30
      IF ( WHEN .LT. CLOCK ) GO TO 40
C FIND INDEX FOR THE NEWLY SCHEDULED EVENT
      TOP = CUREV
 10   IF ( CUR. LE. 0 ) GO TO 20
      IF ( WHEN .LT. EVTIME(CUR) ) GO TO 20
      EVTYPE(TOP) = EVTYPE(CUR)
      EVTIME(TOP) = EVTIME(CUR)
      TOP = CUR
      CUR = CUR - 1
      GO TO 10
C INDEX FOR THE NEWLY SCHEDULED EVENT IS IN TOP
 20   EVTYPE(TOP) = TYPE
      EVTIME(TOP) = WHEN
      RETURN
 30   WRITE (6,901)
      STOP
 40   WRITE (6,902)  CLOCK, TYPE, WHEN
      STOP
 901  FORMAT (' EVENT LIST OVERFLOW' )
 902  FORMAT (' ATTEMPT TO SCHEDULE EVENT IN PAST AT',F10.2/
     &  ' EVENT TYPE',I3/' TO OCCUR AT',F10.2)
      END
```

```
SUBROUTINE ARRIVE

REAL IATIME
COMMON /INPUT/ IATIME

REAL CLOCK
COMMON /TIME/ CLOCK

LOGICAL IDLE
INTEGER NSYS
COMMON /STATE/ IDLE, NSYS

NSYS = NSYS + 1
CALL SCHED( 1, CLOCK + 2.0*URANDM( 0.0 )*IATIME )
ITEMS = 1.0 + 9.0*URANDM( 0.0 )
CALL ENTERQ( ITEMS )
IF ( IDLE )  CALL SERVE
RETURN
END


SUBROUTINE SERVE

REAL CLOCK
COMMON /TIME/ CLOCK

LOGICAL IDLE
INTEGER NSYS
COMMON /STATE/ IDLE, NSYS

IDLE = .FALSE.
CALL DELETQ( ITEMS )
CALL SCHED( 2, CLOCK + ITEMS*URANDM( 0.0 ) + 3.0 )
RETURN
END


SUBROUTINE DEPART

REAL CLOCK
COMMON /TIME/ CLOCK

LOGICAL IDLE
INTEGER NSYS
COMMON /STATE/ IDLE, NSYS

NSYS = NSYS - 1
IDLE = .TRUE.
IF ( NSYS .NE. 0 )  CALL SERVE
RETURN
END
```

2.3 Organization of a simulation program

Typically, a simulation program can be divided into 3 parts:

1. Initialization.
2. Event subroutines.
3. Output routines.

In the initialization part, variables used in the program are set to the appropriate values to reflect the state of the system at simulated time 0. One or more events are also scheduled to start the events going. The main program is concerned with calling the event subroutines at the appropriate simulated times. In order to terminate the simulation, it is necessary to divert control to the output routines. There are three common methods for doing this:

1. When the number of events has exceeded some prespecified amount, call the output routines. This may be done from inside an event routine or from the main program. This technique is appropriate when we want to observe a certain number of events; in the checkout counter system, for example, we might be interested in observing a given number of arrivals (for statistical control, as will be explained in later chapters) and therefore call the output routines from the 'arrive at counter' event routine when the number of arrivals has exceeded the specified amount.
2. Schedule a special event of type 'end of simulation' during initialization with event time equal to the desired termination time. The main program will then call this event routine at the appropriate time and that routine can call the output routines. This technique is appropriate when we are interested in simulating a given amount of real time.
3. When there are no more events in the event list, the main program can call the output routines. This technique is typically used to model systems where there are "hours of business" such that arrivals are shut off at closing time, but customers in the system at closing time are served. In the checkout counter example, we can easily implement this technique by changing the 'arrive at counter' event routine such that it will not do the bootstrap after "closing time", and having the main program check for CUREV = 0.

In some cases, it may be desirable to use a combination of the above techniques. For example, when the same simulation program is to be run many times with widely varying arrival and service rates, techniques 1 and 2 can both be used to terminate the simulation when at least a certain number of events or a certain amount of time has been observed, whichever comes first.

2.4 Data collection

Data for performance measures are collected in the event subroutines. We will use our check-out counter example to illustrate how this can be done. We consider three of the most commmmmmmmmmmmmon performance measures: waiting time, utilization factor, and number of customers in system. A modified program to

gather these performance measures for our checkout counter model appears at the end of this section. The reader may wish to refer to this program listing during the discussions which follow. The mode of termination is as described in technique 2 above: an 'end of simulation' event (WRAPUP) is scheduled during initialization to occur at time 1500.

2.4.1 Waiting time

The waiting time is defined to be the time a customer waits in queue, it is given by:

waiting time = time service begins - time of arrival

The steps required for the computation of waiting times are:

1. When a customer is entered into the queue (in the 'arrive at counter' event), his time of arrival (the value of the variable CLOCK) is saved. An extra argument has been provided to the subroutine ENTERQ for this purpose.
2. When a customer is deleted from the queue (in the 'start service at cashier' event), his time of arrival is retrieved and his waiting time is computed from:

   CLOCK - time of arrival

These steps allow us to compute the waiting time for each customer starting service at the cashier. It is perhaps best to think of the "time of arrival" as being an an attribute of a customer; it is an attribute which we should have added for our own convenience in collecting data for waiting times. We omitted this attribute in the development of the model to avoid complexity. Had we been experienced analysts following the procedure outlined in section 1.5, we would have anticipated the need for this attribute. In some simulation packages, attributes commonly used in gathering statistics are added to each entity by default!

There are several ways to summarize the waiting time data, the most common ones are *mean*, *variance*, and *histogram*. (Of course, these apply equally well to other measures, but we will phrase our examples in terms of waiting time).

Mean and variance:

To collect data for the mean and variance of waiting time, we need the following variables in our example program:

NWAIT - accumulates the total number of waiting times collected
SWAIT - accumulates the sum of waiting times
SQWAIT - accumulates the sum of squares of waiting time

Initially, NWAIT, SWAIT, and SQWAIT are all set to 0. Every time a customer is removed from the queue, a waiting time can be observed and the following steps are added to the program:

```
NWAIT  =  NWAIT + 1
SWAIT  =  SWAIT + waiting time
SQWAIT =  SQWAIT + (waiting time) ** 2
```

At the end of simulation, we calculate the following for output:

```
mean = SWAIT / NWAIT
variance = (SQWAIT - SWAIT ** 2 / NWAIT) / (NWAIT - 1)
```

Intuitively, we interpret the mean as the "best possible guess" for an arriving customer's waiting time (given that we know nothing about the state of the system when he arrives). The variance is an indication of how accurate that guess is (the smaller the variance, the better the guess).

Histogram:

Although our example program does not collect histogram data, we include this discussion here for completeness. If we divide the axis for waiting time into a number of equal length intervals, then a histogram is a plot of the relative frequency of waiting times falling into each interval. Let K+1 be the total number of intervals, and h be the interval length. We define an array W and assign the element W(i), i = 1,2, ..., K, to accumulate the total number of waiting times that fall between (i-1)*h and i*h. The element W(K+1) is used to accumulate the total number of waiting times that are larger than K*h. It is usually wise to provide this "overflow bucket" in case we make a bad choice for K and h. In fact, it would be wise to also keep track of the largest overflow value so that a good choice of K and h can be made with only one more run.

Initially, W(i) = 0 for all i. Every time a waiting time is computed (step 1 in subroutine for 'customer starts service at cashier' event), the interval into which it falls is determined, and the corresponding element of array W is incremented by 1. At the end of simulation, the relative frequencies are obtained by dividing each W(i) by NWAIT (which contains the total number of waiting times collected).

We may generalize the collection of data for a histogram with a third parameter b, to indicate the base value of interest. We need an additional element W(0) in the W array. In this case, we accumulate the total number of waiting times that are less than b into W(0), those in the range (i-1)*h+b to i*h+b into W(i), and those larger than K*h+b into W(K+1). In this way, we have provided both an underflow and an overflow bucket. Again, it is probably wise to record the smallest underflow value in case the choice of K, h and b are poor. We must emphasize here that we cannot in general predict the "best" values for the histogram parameters which will be most effective in illustrating the facets of model behavior, so it is quite often necessary to make a "dry run" of an experiment to arrive at the best values to use throughout the experiment.

It is worthwhile to note that we can compute approximations to the mean and variance of waiting time from the data collected for the histogram. The advantage of this method is that it avoids the necessity of collecting the data for both and it may also avoid truncation error encountered when we accumulate the sum and sum of squares of waiting times. The method is as follows:

```
SWAIT  =  W(1)*(0*h+b) + W(2)*(1*h+b)
          + ... + W(K)*((K-1)*h+b)

SQWAIT =  W(1)*(0*h+b)**2 + W(2)*(1*h+b)**2
          + ... + W(K)*((K-1)*h+b)**2
```

The values of SWAIT and SQWAIT may then be used in the previous formulae to calculate the mean and variance. This method only gives us an estimate because it (a) ignores the underflow and overflow values and (b) assumes that the values of all waiting times falling into an interval are equal to the value of the left end of the interval. The latter problem might be alleviated somewhat by assuming the values to be equal to the mid-point of the interval.

2.4.2 Utilization factor

The *utilization factor* is the fraction of simulated time that the cashier is busy. We thus have:

utilization factor = total busy time / total simulated time

Since the cashier is either busy or idle, we can also write:

utilization factor = 1 - (total idle time / total simulated time)

We have chosen to accumulate the total time that the cashier is busy. The busy times are accumulated in the variable SBUSY. In this simple model, the service times are calculated at the point where the scheduling of the 'depart from system' event occurs; therefore we need only accumulate the service time into SBUSY. In a more complex system, we would have to use variables to remember times at which the server becomes busy or idle and use them to determine the busy and/or idle times.

Since the total simulated time is given by CLOCK, we can get the utilization by:

SBUSY / CLOCK

2.4.3 Number of customers in the system

Mean and variance:

In our supermarket check-out counter example, the number of customers in the system changes when customers are entering or leaving the system. The mean and variance of number in the system are statistics taken over the total simulated time, rather than by making discrete observations (as in the case of waiting times). Thus, if we let T(N) be the total simulated time that the number in the system is N, N = 0,1,2, ..., then the mean number in the system is given by:

(T(0)*0 + T(1)*1 + T(2)*2 + ...) / total simulated time

Note the similarity to the manner in which the histogram frequencies were used to reconstruct the mean in section 2.4.1; however, here we divide by the total simulated time rather than a count of the number of observations because we are multiplying the observed values (i.e., 0, 1, 2, ...) by times rather than by frequencies. The sum of squares is given by:

(T(0)*0**2 + T(1)*1**2 + T(2)*2**2 + ...)

To collect data for the mean and variance, we need the following variables:

```
SIN    -    accumulates  (T(0)*0 + T(1)*1 + ... )
SQIN   -    accumulates  (T(0)*0**2 + T(1)*1**2 + ... )
LASTCH -    accumulates time at which NSYS was last changed
```

Initially, SIN, SQIN, and LASTCH are all set to 0. In the event subroutines, every time *before* NSYS is changed the following calculations are performed:

```
SIN    =    SIN + (CLOCK - LASTCH) * NSYS
SQIN   =    SQIN + (CLOCK - LASTCH) * NSYS ** 2
LASTCH =    CLOCK
```

At the end of the simulation, the total simulated time is given by the value of CLOCK, thus the mean number in the system is given by:

```
mean = SIN / CLOCK
```

and the variance by:

```
variance = SQIN / CLOCK - (mean) ** 2
```

Histogram:

This is a plot of the relative frequency of simulated time that the number in the system is i, i = 0,1,2, ... Data for the histogram can be collected by using an array T and accumulating in T(i) the total simulated time that the number in the system is i.

Initially, T(i) = 0 for all i. Every time before NSYS changes the following calculations are performed:

```
T(NSYS) =    T(NSYS) + (CLOCK - LASTCH)
LASTCH  =    CLOCK
```

At the end of simulation, the relative frequency of simulated time that the number in the system is i is obtained by:

```
P(N) = T(N) / CLOCK
```

Note that the array T we have discussed here is similar to the array W in the discussion of the mean and variance. We can therefore use P(N) to compute the mean and variance of number in system. In particular, we have:

```
mean = P(0)*0 + P(1)*1 + P(2)*2 + ...
```

and

```
variance = (P(0)*0**2 + P(1)*1**2 + P(2)*2**2 + ...) - mean ** 2
```

```
C   MAIN PROGRAM - CHECKOUT COUNTER SIMULATION
      REAL CLOCK
      COMMON /TIME/ CLOCK

      INTEGER CUREV, EVTYPE(10)
      REAL EVTIME(10)
      COMMON /EVENTS/ CUREV, EVTIME, EVTYPE

      INTEGER TYPE

      CALL INIT

10    CLOCK = EVTIME(CUREV)
      TYPE = EVTYPE(CUREV)
      CUREV = CUREV - 1

      IF ( TYPE .EQ. 1 )  CALL ARRIVE
      IF ( TYPE .EQ. 2 )  CALL DEPART
      IF ( TYPE .EQ. 3 )  CALL WRAPUP

      GO TO 10
      END


      SUBROUTINE ARRIVE

      REAL IATIME
      COMMON /INPUT/ IATIME

      REAL CLOCK
      COMMON /TIME/ CLOCK

      LOGICAL IDLE
      INTEGER NSYS
      COMMON /STATE/ IDLE, NSYS

      INTEGER NWAIT
      REAL SWAIT, SQWAIT, SBUSY, SIN, SQIN, LASTCH
      COMMON /STATS/ NWAIT, SWAIT, SQWAIT, SBUSY, SIN, SQIN, LASTCH

      SIN = SIN + (CLOCK - LASTCH)*NSYS
      SQIN = SQIN + (CLOCK - LASTCH)*NSYS**2
      LASTCH = CLOCK
      NSYS = NSYS + 1
      CALL SCHED( 1, CLOCK + 2.0*URANDM( 0.0 )*IATIME )
      ITEMS = 1.0 + 9.0*URANDM( 0.0 )
      CALL ENTERQ( ITEMS, CLOCK )
      IF ( IDLE )  CALL SERVE
      RETURN
      END


      SUBROUTINE SERVE

      REAL CLOCK
      COMMON /TIME/ CLOCK

      LOGICAL IDLE
      INTEGER NSYS
      COMMON /STATE/ IDLE, NSYS

      INTEGER NWAIT
      REAL SWAIT, SQWAIT, SBUSY, SIN, SQIN, LASTCH
      COMMON /STATS/ NWAIT, SWAIT, SQWAIT, SBUSY, SIN, SQIN, LASTCH

      REAL ATIME, STIME
```

```
      IDLE = .FALSE.
      CALL DELETQ( ITEMS, ATIME )
      NWAIT = NWAIT + 1
      SWAIT = SWAIT + (CLOCK - ATIME)
      SQWAIT = SQWAIT + (CLOCK - ATIME)**2

      STIME = ITEMS*URANDM( 0.0 ) + 3.0
      CALL SCHED( 2, CLOCK + STIME )
      SBUSY = SBUSY + STIME
      RETURN
      END


      SUBROUTINE DEPART

      REAL CLOCK
      COMMON /TIME/ CLOCK

      LOGICAL IDLE
      INTEGER NSYS
      COMMON /STATE/ IDLE, NSYS

      INTEGER NWAIT
      REAL SWAIT, SQWAIT, SBUSY, SIN, SQIN, LASTCH
      COMMON /STATS/ NWAIT, SWAIT, SQWAIT, SBUSY, SIN, SQIN, LASTCH

      SIN = SIN + (CLOCK - LASTCH)*NSYS
      SQIN = SQIN + (CLOCK - LASTCH)*NSYS**2
      LASTCH = CLOCK
      NSYS = NSYS - 1
      IDLE = .TRUE.
      IF ( NSYS .NE. 0 )  CALL SERVE
      RETURN
      END


      SUBROUTINE WRAPUP

      INTEGER NWAIT
      REAL SWAIT, SQWAIT, SBUSY, SIN, SQIN, LASTCH
      COMMON /STATS/ NWAIT, SWAIT, SQWAIT, SBUSY, SIN, SQIN, LASTCH

      REAL CLOCK
      COMMON /TIME/ CLOCK

      REAL MEAN, VAR

      MEAN = SWAIT/NWAIT
      VAR = (SQWAIT - SWAIT**2/NWAIT)/(NWAIT-1)
      WRITE (6,901) NWAIT, MEAN, VAR

      MEAN = SBUSY/CLOCK
      WRITE (6,902) MEAN

      MEAN = SIN/CLOCK
      VAR = (SQIN - SIN**2/CLOCK)/CLOCK
      WRITE (6,903) MEAN, VAR

      STOP
  901 FORMAT ( ' WAIT TIME SUMMARY'/
     &         '   NUMBER   =',I10/
     &         '   MEAN     =',F13.2/
     &         '   VARIANCE =',F13.2 )
  902 FORMAT ( /' UTILIZATION FACTOR =',F5.2 )
  903 FORMAT ( /' NUMBER OF CUSTOMERS IN SYSTEM'/
     &         '   MEAN     =',F13.2/
     &         '   VARIANCE =',F13.2 )
      END
```

```
      SUBROUTINE INIT

      REAL CLOCK
      COMMON /TIME/ CLOCK

      INTEGER CUREV, EVTYPE(10)
      REAL EVTIME(10)
      COMMON /EVENTS/ CUREV, EVTIME, EVTYPE

      LOGICAL IDLE
      INTEGER NSYS
      COMMON /STATE/ IDLE, NSYS

      INTEGER FRONT, BACK, NITEMS(100)
      REAL ARRVTM(100)
      COMMON /QUEUE/ FRONT, BACK, NITEMS, ARRVTM

      REAL IATIME
      COMMON /INPUT/ IATIME

      INTEGER NWAIT
      REAL SWAIT, SQWAIT, SBUSY, SIN, SQIN, LASTCH
      COMMON /STATS/ NWAIT, SWAIT, SQWAIT, SBUSY, SIN, SQIN, LASTCH

      CLOCK = 0
      CUREV = 0
      NSYS = 0
      IDLE = .TRUE.

      FRONT = 0
      BACK = 0

      IATIME = 6.0

      NWAIT = 0
      SWAIT = 0
      SQWAIT = 0
      SBUSY = 0
      SIN = 0
      SQIN = 0
      LASTCH = 0

      CALL SCHED( 1, 0.0 )
      CALL SCHED( 3, 1500.0 )
      RETURN
      END
```

2.5 Gathering data by sampling

The methods of data gathering outlined in section 2.4 are called *complete observation* since we collect data at each point where any state variable of interest changes value. Both the exact form of the statements to be inserted and the exact location at which they should be inserted is readily decidable; in fact, it could be done mechanically by a computer program. We will later see that much of this type of data gathering code is inserted automatically by the GPSS and SIMSCRIPT packages.

The fact that data gathering code can be automatically inserted is important because it reduces the likelihood of human error in this task: erroneously omitting one or more of the code inserts could result in believable but incorrect statistics (which is worse than being obviously wrong!). Moreover, the automatically inserted data gathering code does not clutter up the program listing: the analyst is therefore less distracted from the code which implements the model.

Another method of gathering data is to have a special event, say SAMPLE, which is initially scheduled to occur at some non-zero time and which bootstraps itself to occur periodically throughout the simulation. The SAMPLE event "observes" the values of the state variables being measured. This technique has the advantages mentioned above: we do not have to worry about missing a data collection point and the code which implements the model is not cluttered with the data gathering code. However, this method of *sampled observation* is not as accurate as complete observation, since the sampling does not necessarily coincide with state changes. Furthermore, there is a danger that sampling may occur with a periodicity which matches some periodic behavior of the the system, and will therefore give a biased view of the system state. This latter problem may be alleviated by having the SAMPLE event bootstrap itself with randomly spaced intervals rather than fixed intervals.

2.5.1 Sampling the utilization factor

Since the utilization factor is the fraction of time the server is busy, we sample the system state to see how often we catch him busy. Starting with variables BUSY and NSAMP initialized to zero, we perform the following steps in the SAMPLE event:

```
NSAMP = NSAMP + 1
IF ( .NOT. IDLE )  BUSY = BUSY + 1.0
```

Then, in the WRAPUP routine, we calculate:

$$\text{utilization factor} \simeq \text{BUSY} / \text{NSAMP}$$

We may convince ourselves that this is the correct approximation by the following argument: if SAMPLE occurs every Δt time units, then when WRAPUP is called

$$\Delta t * \text{NSAMP} = \text{CLOCK}$$

We assume that each time the cashier is seen to be busy, he has been busy since the last sample event. Hence

$$\Delta t * BUSY \simeq SBUSY$$

where SBUSY is the busy time calculated by complete observation (see section 2.4.2). Likewise, when we find that the cashier is idle, we assume that he has been idle since the last SAMPLE event. Thus

$$SBUSY / CLOCK \simeq (\Delta t * BUSY) / (\Delta t * NSAMP) = BUSY / NSAMP$$

The errors introduced by our two assumptions tend to counterbalance if many samples are taken; in fact, as Δt becomes small, the error vanishes. However, the amount of CPU time spent executing the SAMPLE event will also become greater as Δt becomes small, so there is a trade-off involved.

2.5.2 Sampling the number in system

As in the previous section, we make the assumption that whatever state prevails at the time of the SAMPLE event has been in effect since the last SAMPLE event. We use the variable SYSN, which is initialized to zero, to accumulate samples of the value of NSYS in the SAMPLE event:

```
SYSN = SYSN + NSYS
```

In WRAPUP we may then calculate the mean number in the system as:

$$mean \simeq SYSN / NSAMP$$

where NSAMP is accumulated as in section 2.5.1.

Histogram:

As in section 2.4.3, we use an array T to accumulate the number of samples for which the number in the system is i, i = 0, 1, 2, ... Initially, T(i) = 0 for all i. In the event SAMPLE, we accumulate:

```
T(NSYS) = T(NSYS) + 1.0
```

In WRAPUP, we calculate the approximate relative frequencies as:

$$P(i) \text{ (approxeq) } T(i) / NSAMP \quad \text{for all } i$$

where NSAMP is accumulated as in section 2.5.1.

2.6 Snapshots and resetting

2.6.1 Steady-state and transient behavior

The mean values for utilization factor and number in system as calculated by the methods of sections 2.4 and 2.5 represent a "good guess" at values of state variables. For example, a given utilization factor ($0 \leqslant \rho < 1$) indicates that if we were to randomly select a time at which to observe the system, the probability that the server is busy is ρ. Likewise, knowing from previous experience that the mean number in the system is $\bar{n}$, then $\bar{n}$ will typically be a good guess for the number of customers in the system at our randomly selected observation time.

However, a quick inspection of the program presented in section 2.4 will show that if our randomly selected time happens to fall within the first few units of simulated time, the server is undoubtedly busy and the number of customers in the system is one. This is because an arrival is scheduled for time zero, the server immediately becomes busy and the next customer is bootstrapped to arrive at a later non-zero time. In fact, the system state variables may require a great deal of simulated time to settle to the "average" values from a long run; moreover, we may well wonder if they really ever settle to any sort of representative values at all. We refer to the time during which the system state variables may be expected to assume values whose mean, variance and distribution reflect those gathered from a long period of observation (or predicted by analytical techniques) as the *steady state* of the system. The time during which the state variables are not well described by these measures is referred to as *transient response*. A system cannot reach steady state unless the arrival distributions, service distributions, queueing disciplines, and network topology remain constant. Also, the length of the transient response will depend upon the state of the system at the beginning of the transient, the arrival distributions, service distributions, queueing disciplines and network topology.

The transient response which occurs at the beginning of a simulation is referred to as the *start-up transient*. There have been some analytical results derived for the transient response of simple systems, but very little is known about this phenomenon in precise terms. In qualitative terms we may observe that the transient reponse tends to:

1. increase as $\rho \rightarrow 1$ (i.e., $\lambda \rightarrow \mu$).
2. increase as the complexity of the network increases; for example, if the system is a series of single server queues (as in a cafeteria), each individual server cannot reach steady state until the server preceding him does.
3. depend in complex ways upon other factors such as the exact distributions of interarrival and/or service times and upon the queueing disciplines.
4. May be infinite if $\rho \geqslant 1$ (i.e., $\lambda \geqslant \mu$).

Obviously, our results are affected by the fact that we collect data during the start-up transient; therefore we find it desirable to avoid this if possible.

2.6.2 Locating and avoiding start-up transients

There are three basic methods for negating or avoiding the effects of the start-up transient:

1. run the data gathering so long that the transient reponse is a negligible portion of the the total simulated time.
2. initialize the system state to values which are very near those expected during the steady state, thereby eliminating the transient response.
3. do not accumulate data for statistics until the start-up transient is past.

All three methods have merit: for simple systems (no series connections) method 1 may suffice, while complex systems (having many series connections) may have such long transients that method 2 is necessary; in systems of medium complexity (one or two series connections) method 3 can be less costly in computer time than 1 and easier to manage than 2.

Snapshots:

In all methods outlined above it is necessary to locate the end of the start-up transient. One technique of doing this is to take "snapshots". A *snapshot* is a statistical summary limited to a subinterval of the simulation. This is easily done for our example checkout counter system model using the routine WRAPUP introduced in section 2.4; by changing the STOP statement to a RETURN, having the routine call SCHED to bootstrap itself and changing INIT to schedule the first occurence of this newly created "SNAP" event at an early point in the simulation. The example system is too simple to exhibit transient response convincily unless $\rho \geqslant 1$. If steady state is attained, one will note that the successive outputs from SNAP tend to change by smaller and smaller amounts. It is difficult to quantify what constitutes a small enough change in the outputs to constitute a basis for deciding that the steady state has been attained; the analyst must exercize judgement here. These dry run outputs may then be used to estimate a good initial state, or to decide when data gathering should start.

Resetting:

Once we have located the end of the start-up transient, we would like to ignore data collection until it is past. Of course, it would be rather inconvenient to preface each set of data gathering statements outlined in section 2.4 by an IF statement which checks to see if it is yet time to gather data. The solution to this problem is to *reset* the data collection at the end of the start-up transient; we do this by scheduling an event RESET at that time. The RESET event should reset the variables used to accumulate data as follows:

```
NWAIT = 0
SWAIT = 0.0
SQWAIT = 0.0

SBUSY = 0.0

SIN = 0.0
SQIN = 0.0
LASTCH = CLOCK

RTIME = CLOCK
```

Setting values back to zero destroys all information gathered during the transient. It is most important to note how variables which "remember" times are handled: they are set equal to the current time, so observations using them will ignore time which precedes the beginning of steady-state observation.

Note that we have also provided a new variable, RTIME. This variable is added to the common block /TIME/ and is set to zero in the INIT subroutine. It is used to remember the last time a RESET event occurred. In the WRAPUP event, we now must divide by (CLOCK−RTIME), which is the new period of observation.

Moving window:

We may combine the techniques of snapshots and resets to obtain a *moving window* snapshot. As described above, each snapshot includes data which was used in previous snapshots. However, if the event routine SNAP calls RESET before returning, the next snapshot will summarize data gathered since the last RESET. One problem with this technique is that there will be a transient due to the state the system was left in at the time of the SNAP and RESET; thus each output will be "tainted" by the previous output. This phenomenon is known as *dependence*, and will be discussed in chapter 6. This is much the same problem as the start-up transient, and can be solved by the same technique: the SNAP should bootstrap itself to occur at time

$$\text{CLOCK} + \Delta r + \Delta s$$

and schedule a RESET at time

$$\text{CLOCK} + \Delta r$$

where Δr is large enough for the system to reach a state which is independent of the state at the time of the SNAP. The next SNAP will then summarize the measurement data collected in the previous Δs time units.

CHAPTER 3
Development of Models

In this chapter, we discuss a number of variations of queueing systems one might encounter. Several detailed examples of systems are abstracted into models.

3.1 Some frequently encountered queueing situations

Our examples thus far have concentrated on systems which can be modelled by a single server queue. There are many other possible variations of queueing systems which are encountered frequently in everday life. The following examples are not intended to be an exhaustive list of possibilities; rather, it is a list of some of the more common situations. Neither do the examples represent a firm categorization of possibilities: one may expect to see many systems which have characteristics of two or more of the examples.

3.1.1 Infinite-population models

This is the class of queueing models where the customer interarrival time distribution is independent of the state of the system. It is often used to study service facilities which serve a large population of customers, e.g., bank, supermarket, gasoline station, and post-office. The usual assumption is that the interarrival time is exponentially distributed (this will be explained in later chapters).

3.1.2 Finite-population models

This model is often called the Machine Repairman Model. It represents a finite (and relatively small) number of customers requesting service from a service facility. It is useful for modelling interactive computer systems where customers are interacting with the system through their terminals. The typical behavior of each customer is to go through alternating periods of thinking at the terminal and waiting for the system response. It is intuitively clear that the arrival rate to such a system is a decreasing function of the number in system; and when all customers are in the system, there will not be another arrival until a departure has occurred.

In a machine repairman model, each machine is analogous to a customer at terminal, the time during which a machine is in operation is analogous to the think time, and the time a machine spends in the repair shop is analogous to the system response time.

3.1.3 Multiple servers

In a service facility with multiple servers, customers can form a single FCFS queue, with the person at the head of the queue proceeding to the first available server (often referred to as a *quickline*) or form separate queues in front of each server. The single queue system has the advantage that a customer entering the system before another customer is guaranteed to start service earlier. The variance of the waiting time may also be reduced, since customers do not get trapped behind other customers having long service times, nor are they sometimes fortunate enough to be behind customers with small service times. On the other hand, it has been observed that human servers sometimes slow down with the single queue arrangement, perhaps due to a decreased pressure to empty their own queue.

3.1.4 Jockeying

In a system with multiple servers, customers sometimes discover that they have selected a queue which just happens to have one or more customers with a large service requirement at the head. As a result, other queues become much shorter than their own, so they leave their queue and join a shorter one. This "jockeying for position" will have significant effect on the wait times observed in most systems.

3.1.5 Classes of customers

We may classify customers according to their interarrival time and/or service requirement. A typcial example is the distinction between interactive jobs and batch jobs inside a computer system. In a supermarket, we might divide customers into two classes: those having more than eight items and those with eight or fewer items. With classes of customers, we can implement a priority queueing discipline based on class membership, or apportion different numbers of servers exclusively to each class.

3.1.6 Discriminatory queueing disciplines

This is a generalization of the "classes of customers" situation described above. Very often, it is desirable to give priority to customers with small service requirements. For example, we might have our supermarket cashier select the next customer to be served on the basis of the number of items each customer in the queue has. It is generally true that giving priority to shorter jobs would result in a smaller mean waiting time. In fact, it has been proved that with an exponential interarrival time distribution, the shortest-job-first discipline gives the smallest mean waiting time.

3.1.7 Preemptive disciplines

For some servers, it is possible to preempt the job is service, put the job back to queue, and start service on another job. Such disciplines are often used in queueing systems with priority classes. It is also useful in computer systems where each job is given a quantum of CPU time; if the job is completed before its quantum expires, it simply leaves the system, otherwise it is preempted from service and put back to queue. Common examples of quantum-based preemptive disciplines are Round-Robin and Foreground-Background. Both disciplines are designed to give implicit priority to short jobs.

3.1.8 State dependent service rates

Often we will see a case in which the server is aware of the state of the system and will adjust his service rate accordingly. In particular, he may serve more quickly or refuse to serve big customers during periods of peak demand. Later, in the discussion of a moving head disc system, we will see a case in which each customer's service time depends upon the state (i.e. arm position) in which the previous customer left the system.

3.1.9 Reneging

When queues become too long, we find that there is either no more room for customers to wait or that customers are unwilling to wait for a long time. Customers

will either not join the queue at all or will join for a short time, then leave without service. Either case is referred to as "reneging". In a model for which reneging is felt to be an important factor, one should keep track of the amount of reneging which occurs in order to estimate the trade-off between lost business versus expansion. Note that if one adds a server to avoid the lost business that the increased cost of the server may exceed the gain in business.

3.1.10 Servers with varying capabilities

This may take one of two forms: first, some servers are capable of the same service at a different speed (i.e., an IBM 360/40 versus a 360/75); or, second, some servers may be reserved for "special" service (i.e., cashiers who serve customers with eight items or fewer).

3.1.11 Multi-stage service

In some systems, such as a cafeteria, a customer may require several stages of service. When a customer terminates service at one server and joins the queue for the next stage, he allows the customer behind him to enter service. If there is only a finite waiting space between the servers, it is possible that the customer behind cannot enter service because the queue for the next server has become too long. This phenomenon is known as "blocking"; in a system of this nature, the experimenter should measure the idle time induced in each server by interference from the next server's queue.

We can generalize the case of multi-stage service to a network of connecting queues where customers move from one server to another according to a probability distribution. A common example of this is the *scramble system* cafeteria, where the servers are stationed at separate counters according to the food item provided; customers then proceed in any order to the stations as necessary to get the meal components they want.

3.1.12 Group arrival and group service

Many times we may observe a system in which customers arrive and/or are served in groups. In a restaurant, for example, customers tend to arrive in groups and are served in groups, while in a ferry, customers tend to arrive singly but are served in groups.

3.1.13 Multiple resource requirements

In some systems, we may observe a situation in which more than one "server" is required simultaneously. We have seen an example of this within the Tellers Subsystem of our bank model. In a computer system, jobs may require use of card readers, memory, the central processor, tape drives and/or line printers at various stages of their service. Multiple resource requirements sometime result in a condition known as "deadlock"; this is a state in which two or more customers are holding resources requested by another and requesting resources held by another in such a way that none can proceed. This condition is sometimes referred to as a "circular wait". As an example, if our computer system has a single card reader and a single line printer, we may observe a sequence of events in which one job begins service on the card reader, the second begins service on the line printer. The first then

attempts to begin service on the line printer without releasing the card reader while the second attempts to begin service on the card reader without releasing the line printer. Unless some higher level allocation strategy detects the condition, these two jobs will never leave the system and will prevent other jobs from using their resources. There are of course many solutions to this problem, but they are outside the scope of this course. We refer the interested reader to most good texts on operating systems (Shaw [1974] has a very comprehensive treatment). We simply warn the reader that this problem can develop and that simulation packages such as GPSS and SIMSCRIPT II.5 do not detect it!

3.1.14 Time dependent arrival rates

In some queueing systems, there exist peak hours where customers arrive at a rate higher than normal. Typical examples are a cafeteria at lunch time and a highway during rush hours. We can model such systems by scheduling two events of type "change" during initialization. The first event is to occur at the beginning of the peak hours and it changes the arrival rate to a higher value. The second event is used to change the arrival rate back to normal at the end of the peak hours. If possible, complex time-dependent arrival rates should be avoided. Quite often, it will be adequate to run several simulations, each with a fixed arrival rate, with arrival rates selected to cover the range of interest. If it is necessary to observe the system under changing conditions, it may be interesting to initialize the simulation by filling the queue and observing the length of time required to empty the system with no arrivals. This latter technique might be used to evaluate strategies for clearing traffic quickly when a large sporting event is held.

3.2 Moving head disk

3.2.1 System description

A moving head disk is a device for secondary storage in a computer system. A picture of this device is shown in Figure 3.2. The storage medium is a metal disk coated with magnetic materials, and information is stored in tracks. Each track has a number of sectors, and information transfer between the disk and main memory always starts on a sector boundary. There is a single read/write head which can move towards or away from the centre of the disk. This head must be positioned in the right track before information in this track can be accessed. The disk is rotating at a constant speed, and read/write operation is performed as the desired sector rotates into position under the head.

In Figure 3.2, we have only shown the top surface of a single disk system. In general, both surfaces (top and bottom) of the disk are used, and there is a read/write head per surface. In a multi-disk system, there are a number of disks arranged in a stack, and the tracks at the same radial distance from the centre form a cylinder.

In a computer system, I/O requests are generated when data transfer to the disk is required. This can happen when a user (or system) program requires access to a data file; or in the case of a paging system, a page fault has occurred and the referenced page is to be moved to main memory. The operating system keeps track

of a list of I/O requests to the disk, and these requests are served according to some queueing discipline. Common disciplines are FCFS and Shortest-Seek-Time-First (SSTF). Typically, service is started by a "Start I/O" instruction to the disk, and when the I/O operation is completed, an interrupt is generated.

3.2.2 The model and its implementation

The simulation model developed in this section is for a disk having a single surface. It can easily be generalized to more surfaces. We will model the moving head disk by a single server queue. The disk is the server and the I/O requests are customers. The entities and attributes are:

entities	*attributes*
system	arrival rate of IO requests
disk	number of cylinders number of sectors per track rotational speed current access arm position characteristics of arm movement
I/O request	cylinder number sector number

The sets, events, and activities are similar to those for the supermarket check-out counter example in Chapter 2.

We need to characterize the service time for our model. In a moving head disk, the service time is given by the sum of three components: the seek time, the rotational delay, and the data transfer time. The seek time is the time required to position the access arm to the requested cylinder, the rotational delay is the waiting time for the requested sector to move under the head, and the data transfer time is the time required to do the actual data transfer. We assume, for simplicity, that a sector is the unit of data transfer between main memory and the disk. Let ROTSPD be the rotational speed in rev/sec, and NSEC be the number of sectors per track. The data transfer time is then given by:

$$\text{data transfer time} = 1 \;/\; (\text{ROTSPD} * \text{NSEC})$$

The seek time is a function of the number of cylinders moved by the head. Let ARMPOS be the current position of the access arm, the number of cylinders moved is given by:

$$\text{NMOVE} = |\text{ARMPOS} - \text{cylinder number of request}|$$

Note that ARMPOS is the cylinder number of the last request. When the seek portion of the service is complete, ARMPOS will be equal to the cylinder number of the current request. Some possible assumptions about the distribution of cylinders are:

(a) The cylinder number requested is uniformly distributed over all the cylinders. Let NCYL be the total number of cylinders, then

$$\Pr(\text{cylinder number requested} = i) = 1/NCYL \quad \text{for } i = 1,2,\ldots, NCYL.$$

(b) $\Pr(\text{cylinder number requested} = i) = p(i)$, where $0 \leq p(i) \leq 1$, and $p(1)+p(2)+\ldots+p(NCYL) = 1$. This assumption allows the representation of some cylinders being requested more often than others.

(c) $\Pr(\text{cylinder number of next request} = i \text{ conditioned on cylinder number of last request} = j) = p(j,i)$, where $p(j,1)+p(j,2)+\ldots+p(j,NCYL) = 1$ for each j. This assumption allows some dependency between consecutive requests.

We now characterize the functional relationship between the seek time and the number of cylinders moved. A typical plot is shown in Figure 3.2. The non-linear nature of the seek time is due to the acceleration and deceleration of the arm. The most accurate method is to store this relationship in a table. This method requires NCYL storage locations. To save storage space, one can use the following simplifying assumption:

$$\begin{aligned} \text{seek time} &= 0 && \text{if NMOVE} = 0 \\ &= C1 + C2*NMOVE && \text{if NMOVE} > 0 \end{aligned}$$

where C1 and C2 are constants selected to make a reasonable linear approximation to the actual arm move characteristic.

To determine the rotational delay, we need to know the position of the read/write head (in terms of sector number) at some reference point in simulated time. Without loss of generality, we can assume that at simulated time 0, the head is at the beginning of sector 1. At any time in the simulation, we can use the rotational speed to determine the exact position of the head; and then use this information to calculate the rotational delay.

This method requires us to characterize the sector number of each request. We can use assumptions similar to those listed above for the cylinder number.

A less complicated method is to assume that the rotational dealy is uniformly distributed between 0 and one revolution. This assumption implies that the sector requested is equally likely to be anywhere between 0 and 1 revolution away from the head.

3.2.3 Input parameters and performance measures

Suppose our objective is to compare the performance of the FCFS and the Shortest-Seek-Time-First (SSTF) discipline. SSTF is a discipline that gives priority to the request that is closest to the read/write head in terms of cylinder number. We need to decide on our model assumptions; from the discussions in the last section, we see that there are a number of alternatives. A possible (and perhaps simplest) set of assumptions is listed below:

(a) Interarrival time is uniformally distributed between 0 and $2/\lambda$ (λ = mean arrival rate).

(b) Cylinder number requested is uniformly distributed over all cylinders.

(c) The seek time is characterized by the linear function described above.

(d) Rotational delay is uniformly distributed between 0 and one revolution.

It is important to point out that these assumptions reflect the simplifications that we are making in our abstraction, and the accuracy of our model is affected by the assumptions used. In particular, actual experience with computing systems has shown that each of the above assumptions may be invalid enough to lead to erroneous conclusions [LYNC 74]. However, by careful design, these assumptions are easily removed and the model adapted to more realistic assumptions when the time is appropriate. Until then, the simple assumptions have at least the advantage of making the model susceptible to analysis [PINK 73], so the results from the program can be compared with the analytical results to check the implementation of the model.

In the simulation runs, the input parameters are: (a) the mean arrival rate (λ), (b) the constants C1 and C2 for the seek time, (c) number of cylinders, (d) number of sectors and (e) rotational speed. Note that the mean arrival rate is used as an input parameter instead of the mean interarrival time. The reason is that the mean arrival rate is a better parameter to show the load on the system. In the simulation program, however, it is more convenient to use the mean interarrival time.

The basic performance measures are (a) mean service time and (b) mean waiting time. The data collection and final calculations for these are performed as outlined in sections 2.4 and 2.5. The mean service time is included because it is a function of the seek time, which depends upon the location of the previous request; thus, there is an interaction between successive requests which can cause some interesting service time phenomena. We expect SSTF to give better mean service time performance because it is designed to reduce the seek time. In fact, the service time decreases as the arrival rate increases. This is due to the fact that at heavy load, SSTF is more likely to find an I/O request with a cylinder number close to the current arm position. Using FCFS, the requests are served in the same order independent of the load, so the service time does not change with the load.

For mean waiting time, we expect similar characteristics as the mean service time, namely, SSTF will perform better, especially at heavy load. However, SSTF also has the problem of introducing high variance to the waiting time because the middle cylinders are served more frequently than the outer and inner cylinders. An interesting way of demonstrating this is to make very large capacity queues in the simulation program, then run the program several times to find the arrival rate which will cause the queues to overflow after a long time (i.e., the server would be slowly falling behind). When the queue overflow ultimately occurs, note the distribution of cylinder numbers for requests remaining in the queue; for FCFS, they should be uniformly distributed, while for SSTF, there should be a depression in the distribution near the point the arm was positioned at the time of the overflow. This small experiment gives graphic evidence that although SSTF does speed up service times, it does so by penalizing some requests with longer wait times. One danger of this phenomenon is that the unwary experimenter may get the impression that the variance is decreased if the program is run at or above saturation, because the penalty becomes so great that long wait times do not actually occur (they just sit in the queue) and therefore are not incorporated into the data collection.

3.3 A hypothetical bank

3.3.1 System description

The system under consideration is a bank. It can be divided into two subsystems: *table* and *tellers*. The Table Subsystem provides room for customers to fill out deposit (or withdrawal) slips, and the Tellers Subsystem performs the actual transactions. Typically, when a customer enters the bank, he first goes to the table to fill out a slip, and then goes to the tellers. It is also possible that this customer goes directly to the tellers without using the table.

Both subsystems can be considered as service facilities with multiple servers. In the Table Subsystem, the number of servers is determined by the number of customers that can use the table simultaneously. This number is dependent on the size of the table and on how the customers are positioned around the table. In the Tellers Subsystem, the number of servers is the same as the number of tellers; that is, each teller can serve only one customer at a time.

The queueing discipline in the Table Subsystem is not easy to characterize exactly because customers waiting for room on the table simply move around until they find room. In the Tellers Subsystem, a single queue is formed, and the queueing discipline is FCFS.

When a customer leaves the Table Subsystem, he enters the Tellers Subsystem immediately. When a customer leaves the Tellers Subsystem, he also leaves the bank.

3.3.3 The model and its implementation

The entities and attributes for our hypothetical bank are given by the following table:

entities	*attributes*
system	number of tables number of tellers arrival rate of customers
teller	skill
table	size
customer	time to fill out deposit slip service requirement at teller

There are two sets in this system, the owner and member entities are:

set	*owner*	*members*
table queue	table	customer
teller queue	tellers	customer

We can list the events and activities separately for the Table Subsystem and the Tellers Subsystem. In particular, we have:

Table Subsystem:

events	*activities*
customer arrives at table	
	customer waits for room at table
customer starts filling out deposit slip	
	customer fills out deposit slip
customer leaves table (arrives at Tellers Subsystem)	

Tellers Subsystem:

events	*activities*
customer arrives at teller	
	customer waits for service
customer starts service at teller	
	customer receives service from teller
customer departs from system	

3.3.3 Detailed model for customer receiving service

We now refine our model to include a detailed description of the service provided by a teller. Typically, he spends some time to go over the deposit slip, check the signature, etc., and then uses a terminal to update the balance in the passbook. After using the terminal, he returns the passbook to the customer, and in case of a withdrawal, he also pays the customer the appropriate amount of money. Since there are usually fewer terminals than tellers, a teller may have to wait for the terminal he is assigned to to become free (note: for easier accountability, tellers typically perform all transactions at the same terminal).

The service provided by the teller can therefore be divided into 3 parts:

part 1: Service time before teller uses terminal.
part 2: Waiting time and service time for teller at terminal.
part 3: Service time after teller uses terminal.

To include the terminals into our model, we need the following new entities and attributes:

entities	*attributes*
system	number of terminals
teller	assigned terminal
terminal	response time
customer	part 1 service requirement part 2 service requirement part 3 service requirement

We also need a new set for the terminals:

set	*owner*	*members*
terminal queue	terminals	teller

Finally, we have the following new events and activities for the service provided by the teller:

events	*activities*
customer starts service at teller	
	customer receives part 1 service
teller arrives at terminal	
	teller waits for terminal
teller starts part 2 service	
	teller performs part 2 service
teller departs terminal	
	customer receives part 3 service
customer departs from system	

Note that we have replaced the activity "customer receives service from teller" in the less detailed model by a sequence of events and activities. This represents a top-down approach to model development. As more detail is desired, we can continue this top-down development by replacing activities with more complex subsystem interactions. As a further example of this, consider that we have abstracted the service provided by the terminals into a single attribute called "response time". We can go one step further in our top-down analysis and model the behavior of the computer system with which the terminals are interacting. This detailed model for the computer system will characterize the system response time.

Although this system is an example of the case of multiple resource requirements wherein we cautioned against the possibilities of deadlock, we note that in this system deadlock cannot occur because the sequence of events and activities for any combination of customers cannot interact in any way which will cause any terminal/teller pair to block on a "circular wait" condition.

3.3.4 Input parameters and performance measures

Suppose our objective is to predict system performance as a function of the number of tellers and/or number of terminals. We can make some preliminary observations before actually running the simulation. It is rather obvious that increasing the number of tellers would tend to decrease the customer's waiting time for the tellers. However, if the number of terminals is not increased as well, the tellers may have to wait longer for a terminal. This is due to an increase in contention for terminals within the Tellers Subsystem.

It is also obvious that increasing the number of terminals would tend to decrease a teller's waiting time for a terminal. The customer's total waiting time is also expected to decrease because the total time a customer spent with the teller has been reduced. It should be noted, however, that in multiple server queueing systems, there is usually a decreasing rate of return (in terms of improvement in waiting time) as we

provide more servers. From the management point of view, it is important to determine the best combination of tellers and terminals so that the cost to the bank is not excessive, and the customer's waiting time is within some tolerable limit.

To design a simulation model for the above objective, we don't need to include the Table Subsystem. The input parameters are:

(a) Customer interarrival time.
(b) Part 1 service time.
(c) Part 2 service time.
(d) Part 3 service time.
(e) Number of tellers.
(f) Number of terminals.

In the simulation runs, parameters (a) to (d) are assumed to be fixed, and data for performance measures are collected for different combinations of tellers and terminals. The performance measures of interest are:

(a) Customer's service time at teller.
(b) Teller's waiting time at terminal.
(c) % of time each teller is busy.
(d) % of time each terminal is busy.

An interesting way to deomnstrate the effect of additional terminals would be to plot the ratio of customer's service time at teller to teller's waiting time at terminal versus the number of terminals.

Chapter 6

Statistical Tests

6.1 Terminology

Now that we have developed some techniques for building simulation models, we must consider how to make use of the results. Generally, we will want to do two things:

1. Show that two results are *the same* - for example, we would like to convince people that the results from our model are representative of the real system (this is known as validation), so we compare the performance measures output by our model with those obtained by observing the real system.
2. Show that two results are *different* - for example, when we make modifications to the model, we would like to see if the performance measures are affected.

It should be obvious that simply comparing two results will not be convincing; results may look "the same" or "different", but we cannot be sure that this is not simply a chance happening. This becomes apparent when the same simulation program is executed several times, each time with a different seed for the pseudorandom number generator (see chapter 4). The performance measures will differ, even though the different sequences of pseudorandom numbers have essentially the same properties. The different arrangement of arrivals and departures caused by the differing sequence perturbs our performance measures in such a way that our conclusion could be erroneous if we were to rely on the results of a single execution. Thus, we must rely on a well-organized plan of executing the various versions of our simulation program and evaluating the results of these executions. This *well-organized plan* is the heart of a *simulation study*. For the purpose of discussing the techniques used in this important phase of a simulation study, we introduce the following terminology:

run: a single execution of a simulation program, or, the output from one moving window snapshot (see section 2.6.2).

factor: a difference in one or more input parameters and/or in the logic of the simulation program (e.g., the logic that implements a system design strategy) which *may* produce observably different values of the performance measures.

experiment: several runs, each with the same factors, but using a different subsequence from the same psuedorandom number generator.

study: two or more experiments with varying factors, but with similar performance measures for statistical comparison.

observation: a single value of a performance measure resulting from a run.

sample: the set of observations of a performance measure gathered from the runs of an experiment.

In the discussions which follow, we will assume that observations are

independent; that is, there is no extraneous factor which will contribute to making our observations in some way directly or inversely dependent upon each other. This problem was mentioned in section 2.6.1, where we warned that outputs from successive moving window snapshots could be dependent; this problem was avoided by separating each snapshot with a period during which data collection was ignored. Note that we may use each moving window snapshot as a run: this avoids the overhead of submitting the program for execution many times.

6.2 Calculation of confidence intervals

Let $x_1, x_2, \ldots, x_n$ be independent observations of a performance measure X from an experiment. If we could make an infinite number of observations of X, or if an analytical solution is available, we could know the *theoretical mean*, E(X), and the *theoretical variance*, VAR(X) of X. However, we must limit ourselves to a finite number of runs for practical reasons, and often no analytical solution will be known. Given an experiment consisting of n runs, we may calculate the *sample mean* as:

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

This statistic is said to have n-1 *degrees of freedom* because if $\bar{x}$ and the value of n-1 of the x_i's are known, the value of the remaining x_i can be calculated. With less information, this cannot be done, hence the degrees of freedom of a statistic is a measure of its "information content".

From the observed x_i's, we can also calculate the *unbiased variance*:

$$\sigma^2 = \frac{\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2/n}{n - 1}$$

Assuming that the n observations are from a normal distribution, we may calculate the probability of a given $\bar{x}$ if we know E(X). The procedure is:

1. Calculate $\bar{x}$ and σ^2.
2. Let $t_s = \frac{|\bar{x} - E(X)|}{\sqrt{\sigma^2 / n}}$
3. Look in a table of Student's t distribution in the row for n-1 degrees of freedom.
4. Look across the row until the largest value less than or equal to t_s from step 2 is found.
5. At the top of the column found in step 4 is a probability which indicates how frequently this $\bar{x}$ (and more extreme values of $\bar{x}$ from E(X)) would occur given that our assumptions and sampling techniques are correct.

The value obtained from this procedure is called the *significance level* (or significance), and is the probability of such a result occurring by chance alone. We will indicate the significance level with the symbol α.

The main problem with the above procedure is that one usually cannot know E(X), so it is of very limited use. It is possible, however, to solve for E(X) and calculate a *confidence interval* which allows us to say: "the interval (a,b) contains E(X) with probability (or confidence) c". Note that the concept of confidence and significance are related:

$$\alpha = 1 - c \text{ and } c = 1 - \alpha$$

We illustrate the use of a confidence interval by the following example:

Suppose we have 17 observations with $\bar{x} = 9.24$ and $\sqrt{\sigma^2 / 17} = 2.0$. If we desire confidence 0.95, or significance 0.05, then from the table of t, row 16, column 0.05, we obtain the value 2.12. This means that

$$(|19.24 - E(X)|) / 2 \leqslant 2.12$$

with confidence 0.95, which is the same as:

$$|19.24 - E(X)| \leqslant 4.24$$

or

$$-4.24 \leqslant 9.24 - E(X) \leqslant 4.24$$

or

$$-13.48 \leqslant -E(X) \leqslant -5.0$$

or

$$5.0 \leqslant E(X) \leqslant 13.48$$

with confidence 0.95. There are two alternatives for reducing this range. First, we could reduce our confidence requirements, or second, assuming that the confidence must be maintained, we may try to make more runs or increase the length of each run. Increasing the number of runs should result in a smaller value for $\sqrt{\sigma^2 / n}$, which would decrease the range; longer runs typically reduce the variance σ^2, which will also decrease the range. The amount of computer time required for an experiment is roughly proportional to the number of runs times the length of the runs, so there may be an optimum choice for the number and length of runs for a given experiment; that is, we may find that increasing the number of runs and proportionally decreasing the length if the runs (or vice-versa) will yield a smaller confidence interval. Unfortunately, there is no way to know in advance what the best choice is. In a large simulation study, it may be worthwhile to conduct some "mini studies" to determine the best trade-off between number and length of runs.

In summary, we calculate the confidence interval for confidence c as:

$$\bar{x} - T(\alpha,n-1)\cdot\sqrt{\sigma^2 / n} \leqslant E(X) \leqslant \bar{x} + T(\alpha,n-1)\cdot\sqrt{\sigma^2 / n}$$

where $\alpha = 1 - c$ and $T(\alpha,n-1)$ is the value from the Student's t table for n-1 degrees of freedom and significance α (confidence c).

6.3 Use of the null hypothesis

The above procedure may be generalized to obtain a measure of "how different" two experimental results are. This difference will of course be phrased as the probability of observing results that are different under certain assumptions. The major assumption of interest is that the two experiments being compared consist of samples drawn independently from the same normally distributed random variable; that is, they should estimate the same E(X) and VAR(X) and should approximate the same normal distribution.

In the interest of scientific honesty, it is important that we proceed according to a method which will prevent improper conclusions. We begin by phrasing a *null hypothesis* about the distribution of the observations in the two samples. We also choose a level of significance α which we feel appropriate for our application. Depending on the null hypothesis, we then compute a statistic and compare it with the entry in a table. If our statistic is less than the one in the table, we conclude that our observations do not show evidence against the null hypothesis. On the other hand, if our computed statistic is greater than or equal to the entry in the table, we can only reject the null hypothesis. This rejection could be due to many factors:

1. The two experiments sample from normally distributed random variables X_1 and X_2 where $E(X_1) \neq E(X_2)$, but $VAR(X_1) = VAR(X_2)$.
2. One or both experiments sample from non-normal distributions.
3. The two experiments sample from random variables X_1 and X_2 where $VAR(X_1) \neq VAR(X_2)$.
4. Any combination of the above.
5. Chance (with probability α we made an erroneous decision)

It is important to note that rejection of the null hypothesis is completely objective; it does not say, for example: "the means differ significantly because...". Ultimately, we would like to attribute rejection of the null hypothesis to some factor which differed between the two experiments; perhaps a different queueing discipline was used, or another server was added to the model, or.... But these arguments must be subjective, with the statistical tests serving as *circumstantial evidence.*

6.4 Testing the equality of variances (F test)

It is possible that although the runs of our experiments are samples from normally distributed random variables, the two random variables have different variances. We may test to see if the variances differ significantly using the *F ratio*; our null hypothesis is: the two experiments are sampling from normally

distributed random variables having the same variance (though the means may differ).

Given two experiments, consisting of n_1 and n_2 runs, respectively, we calculate the means, $\bar{x}_1$ and $\bar{x}_2$, and unbiased variances, σ_1^2 and σ_2^2, as in section 6.2; for convenience in the use of tables, we renumber the experiments (if necessary) such that $\sigma_1^2 > \sigma_2^2$. We may then calculate:

$$F_e = \sigma_1^2 / \sigma_2^2 \geqslant 1$$

If we desire a level of significance α, we look in the F table for level α, and compare the computed F_e with the F value in column n_1-1 and row n_2-1 of this table. We reject the null hypothesis if the computed F ratio is larger than the value found in the table.

6.5 Testing the difference between experiment means (t test)

If we conduct two experiments consisting of n_1 and n_2 runs respectively, and then calculate the mean and unbiased variance of a performance measure for each (as in section 6.2) to obtain $\bar{x}_1$, σ_1^2, $\bar{x}_2$ and σ_2^2, we may compute confidence intervals for the two as in section 6.2; intuitively, the greater the difference between the two experiments, the less overlap there will be between the two confidence intervals. In fact, it may be necessary to use a very small confidence to have any overlap at all; this is an informal measure of how different the outcomes of the two experiments are. This intuitive notion is formalized by adapting the t test of section 6.2 to test the difference between experimental means. Here, we compute a value of t_s as:

$$t_s = |\bar{x}_1 - \bar{x}_2| / \sigma_{12}$$

where σ_{12} is computed on the basis of the outcome of the F ratio test described in section 6.4.

Case 1: $VAR(X_1) = VAR(X_2)$

If the null hypothesis of the F ratio test was accepted, then use:

$$\sigma_{12} = \sqrt{((n_1-1)\cdot\sigma_1^2 + (n_2-1)\cdot\sigma_2^2)/(n_1+n_2-2)} \cdot \sqrt{1/n_1 + 1/n_2}$$

Should the value of t_s be greater than or equal to the entry in the Student's t table at row n_1+n_2-2 (the pooled degrees of freedom for the two experiments) and in the column α (where α is the chosen level of significance), then we reject the null hypothesis which states that the two experiments drew samples from normally distributed random variables with the same mean and variance.

Case 2: $VAR(X_1) \neq VAR(X_2)$

If the null hypothesis of the F ratio test was rejected, then use:

$$\sigma_{12} = \sqrt{\sigma_1^2/n_1 + \sigma_2^2/n_2}$$

The t test in this case will not be exact. A correction factor for the differences in variances must be used. We do this by computing a new value of the t distribution to test against rather than using the values directly from the t table. This corrected value is:

$$T(\alpha) = \frac{T(\alpha,n_1\text{-}1)\cdot\sigma_1^2/n_1 + T(\alpha,n_2\text{-}1)\cdot\sigma_2^2/n_2}{\sigma_1^2/n_1 + \sigma_2^2/n_2}$$

where $T(\alpha,n\text{-}1)$ is the value from the Student's t table for n-1 degrees of freedom and significance α. If the computed $t_s \geqslant T(\alpha)$, we reject the null hypothesis which states that two experiments drew samples from normally distributed random variables with the same mean but different variances.

We can get an intuitive feel for this technique by considering the case for $n_1=n_2$. The above correction then reduces to $T(\alpha) = T(\alpha,n_1\text{-}1) = T(\alpha,n_2\text{-}1)$. Thus, the correction has allowed us only n_1-1 degrees of freedom rather than the n_1+n_2-2 which would have been allowed if $VAR(X_1) = VAR(X_2)$.