As a tool for quantitative investigation, digital simulation is especially suited to the study of stochastic processes having many interdependent variables. Not only can a simulation model be modified to reflect structural changes in a process, but it can be used to gain insights during the design of the process. These properties recommend the use of digital simulators in the design of complex teleprocessing systems.

This paper comments on the main considerations involved in choosing between general-purpose and special-purpose simulators.

On teleprocessing system design

Part VI The role of digital simulation

by P. H. Seaman

Common to all teleprocessing applications is the need to link remote terminal points to a centrally located computer and files. In a teleprocessing system, transactions may be entered into the system at any time and processed as they occur, rather than being batched for entry at some pre-scheduled time. In such a system, a typical transaction is the entry of data at a terminal, transmission and processing of the data against the files, and completion of the action by transmission back to the terminal.

In implementing such a system, many engineering compromises are necessary. Equipment reliability, speed, and capacity must be weighed against cost, with the final balance often leading to time-sharing of equipment. Typically, several terminal points must share one communication line; hence, if one terminal is transmitting, others may have to wait. In the systems center, files may share single-access disk storage units; as a result, if one access is in progress, other access requests must wait in queues. Consequently, in designing these systems, unusual interest is attached to estimates of the following:

- The quantitative delays that result from queuing of requests for time-shared facilities
- The amount of storage needed to hold the queues

Delays are to be compared with customer requirements, whereas storage requirements are to be matched with system capacity.

Probability analysis and its offspring, queuing theory, can provide helpful answers to some questions. To a large extent,

however, the theoretical approach requires a simplification of the system being considered. In many cases, especially in the early design stages, such an approach is good because it focuses attention on basic parameters and away from distracting details. But as a design progresses, questions come up that cannot be answered by queuing models. For instance, the logic of the system may entail decisions that depend, in complex ways, upon the state of the system at the time the decisions are made. Examples might involve recovery from exhaustion of a main-storage area, or the flow of data around a network of dependent queues (the output of each queue feeding other queues).

Our objective is to discuss questions that arise in studies involving digital simulators. How can simulation help me? Should I write a new simulation program or use one already available? What forms do simulators take? How are the results of a simulator to be interpreted?

Considerations in choosing simulation

Although a simulation study should never be lightly undertaken, such a study can provide information that cannot be obtained by analytical methods. The advantage of simulation is that intricate detail concerning the logic of system operation can be incorporated, and the effects of changes in this detail can be evaluated. As long as the requisite operating detail is not available, one should be satisfied with the results of an approximate algorithmic technique that does not reflect all structural detail.

Very often, however, data not available at the beginning of a study are expected to become available during the course of investigation. In such cases, there are two arguments for beginning simulation at an early stage. First, the basic structure of the early model outlines broad areas of system operation (where fine details have little impact) and provides a correlation with previous mathematical calculations. Second, the early model provides an operating framework into which detail may be incorporated when such becomes available. The design procedure is rarely so clear-cut that one can serially (1) define the problem, (2) collect statistics and write a model, and (3) obtain results to solve the problem. Usually the procedure is iterative and quite exploratory. Often several simple simulations may be helpful in bringing the goals of further simulation into focus. In fact, building pilot models with hypothetical data may yield insights into the large-scale trends in the problem. Just as important, this early exercise builds confidence in the mechanics of simulation itself, and does so while the designer is still free of the multitude of distracting detail that soon engulfs a system study. Too often, the designer becomes so concerned with details at an early stage that he does not acquire sufficient understanding of the potential of his simulator program.

On the other hand, there is a tendency to overrate the value of early output simply because it comes from a simulation. One

must be careful to qualify extrapolations from a preliminary model. As model building proceeds and more detail is incorporated into the model, the results begin to more closely reflect the special nuances of the particular system being designed. It then becomes meaningful to ask questions about minor alternatives of design. A model should grow; it should be viewed as the basis for making design decisions as the need for them occurs. After a system is completed, the model is still useful in evaluating the changes that inevitably occur during the operational period of the system.

As a result of simulation's ability to deal with many details, it is a good tool for studying extensive and complicated computer systems. With simulation, one may assess the interaction of several subsystems, the performances of which are modified by internal feedback loops among the subsystems. For instance, in a teleprocessing system where programs are being read from drum storage and held temporarily in main storage, the number of messages in the processing unit depends upon the drum response time, which depends upon the drum access rate, which, in turn, depends upon the number of messages in main storage. In this case, only a system-wide simulation that includes communication lines, processing unit, and I/O subsystems will determine the impact of varying program priorities on main-storage usage. Studies of this nature can become very time consuming unless parameter selections and variations are carefully limited. It is no small problem to determine which are the major variations that affect the system. In this aspect, simulation is not as convenient as algorithmic methods with which many variations can be tabulated quickly and cheaply.

It is well to emphasize that simulation is not a tool for synthesis; it does not choose its own variations during a run and come up with an optimum solution. Minor adaptive properties have been built into some simulation programs, but they consist of simple addition of capacity or changes in load. In general, simulation is restricted to determining how a particular configuration will react in a particular environment. It is still very much the designer's function to analyze the results and decide where and how a system may be improved. Thus, the human designer is still the feedback element in the design loop.

Considerations in choosing a simulation system

The choice of a simulation system must be made early in a study. It is quite possible that initial pilot runs with one programmed simulator will disclose difficulties that may be averted in an alternate program. The factors to consider in choosing a simulator are as follows:

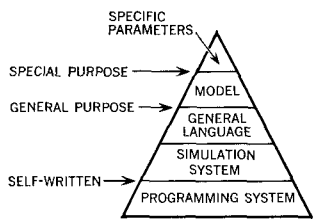
- Time available to make the study
- Programming experience of the user
- Data available
- Size of model to be simulated

- Amount of detail required
- Model characteristics most required
- Usage of the simulator (single or multiple study)
- Length of runs and model running times
- Amount of change and experimentation required

Available simulation programs fall somewhere in the following spectrum:

user-written ⟶ general purpose ⟶ special purpose

Figure 1 Structure of simulation model



The three categories shown refer to the base of the simulator. Once the base structure has been programmed (externally or internally) to model a particular system, the simulator becomes a special-purpose program. It is the amount of modeling the designer must prepare that determines a simulator's classification. This can be seen in the pyramid of Figure 1, showing the parts of a complete simulation model.

A user-written simulator starts from a basic programming system, whereas a special-purpose simulator includes a particular model so that a user need specify only a few parameters. General-purpose simulators lie in between the two, including a simulation system and a general language in their base. The simulation system includes such routines as random-number generators, timing routines, statistical sampling techniques, and report generators. The language may range from abstract generalities covering a wide class of systems to specific objectives aimed at a special class of systems.

Anyone who chooses to write his own simulation program must be presumed to have very compelling reasons for doing so. Among these might be:

- Desired model characteristics are neither available nor feasible in existing programs
- Special input or output features are required. (Modifications to existing programs should always be considered.)
- The program will be used repeatedly, so that the gain in speed through program efficiency can cancel out an investment in initial programming.

The principal advantage of user-written simulators is efficiency. By capitalizing on the peculiarities of the particular application, a special-purpose simulator may be designed to run many times faster than its general-purpose cousin, and to use less computer storage. Since the user-written program is written with cognizance of the system being modeled, its language may well be easier and more natural to use than a general-purpose language. On the other hand, a price must be paid in the initial effort expended in developing the program, and if the program is inflexible in its use, developmental costs must be absorbed by a few applications. Of course, all these considerations may be negated by the simple fact that there isn't any other satisfactory way to do the study.

Special-purpose simulators, on the other hand, are the quickest and simplest to use, since the logic of the system being studied is already more or less completely modeled, leaving only system parameters for the user to specify. Special-purpose simulators are of two types. Those of the first type are written from the beginning in machine language and run very efficiently. However, inasmuch as one must grapple with machine code, changes are often difficult to incorporate. The special-purpose simulators of the second type are built up from a general-purpose base. These are apt to run more slowly and make less-efficient use of storage than the first type. The advantage of a simulator of the second type is the ease with which it can be modified, since under its special-purpose facade it still employs a general language. The utility of a general-purpose program in a specialized study area is greatly extended if there are several special-purpose models of the second type available. The latter can incorporate the most advanced techniques developed by constant users of the general language and make them available to the transient user, either to adopt as they stand, or to serve as prototypes in a special modification.

For a majority of system studies, the choice of a simulation vehicle will be one of the available general-purpose programs. With their ready-made simulation languages, these free the user from the burden of computer details in the same way that FORTRAN provides the analyst with an algebraic language. This reduces initial programming time and relieves the need for experienced programmers. However, the non-trivial problem of expressing the model in the simulator language may still require several man-months in some cases.

Two well-known simulation languages suitable for teleprocessing system simulation are found in SIMSCRIPT and GPSS (General Purpose Simulation System).[1–4] Of the two, SIMSCRIPT is the most abstract, permitting a wide diversity of application structures at some expense in speed of model building. GPSS affords more of a compromise between generality and ease of use. Because it assumes certain structural features in its basic organization, the range of systems it handles efficiently is smaller than is the case with SIMSCRIPT.

A number of special-purpose simulators, some of which appear to be applicable to teleprocessing system design, have been mentioned in the literature. For various reasons, simulators in this class tend to be considered proprietary by computer vendors and consulting firms, and documentation is therefore not publicly available. Those who sense the need for a special-purpose simulator should seek out information from vendors and consultants. In any case, the choice of a special-purpose simulator should be made with the technical capabilities, limitations, and decision-making logic of the simulator clearly in mind. An investment in simulation results is an investment in confusion unless the result-generating mechanism is clearly understood by the user.

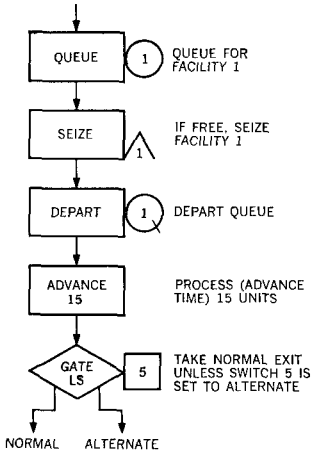
To make the discussion reasonably concrete, we will employ

the widely available GPSS as a reference point in remarking upon some of the most necessary and desirable properties that belong in special-purpose simulators for modeling teleprocessing systems. A brief recapitulation of GPSS terminology may reduce the need for readers to consult the GPSS references.

GPSS

The basic building units employed by GPSS, generally called *entities*, are of four types: facilities, stores, switche, and transactions. A *facility* can perform only one function at a time, and may represent objects such as machines or service counters. As a space-sharing facility, a *store* may hold many objects at once, and represent structures such as a parking lot or inventory in stock. A *switch* is a two-state device that may either block or divert a flow, as would a traffic light or a detour sign. A *transaction* is a discrete unit of traffic that interacts by utilizing facilities, entering stores, and being gated by switches. It may represent orders in process, automobiles, customers, or messages.

Figure 2 Example of GPSS block diagram



To structure entities and define a logical flow of transactions, the GPSS language contains basic operations such as "seize facility," "enter storage," and "set switch." Each operation is called a *block*, and a network of operations is called a *block diagram*. The correspondence to ordinary block diagrams or flowcharts is deliberate. Transactions are caused by the GPSS program to "flow" through the diagram from block to block, automatically following the arrows and executing the operations as they are encountered. (See Figure 2.)

The GPSS program provides various supporting service routines. An input assembler translates the block cards and sets up an internal model. Sampling and output routines can automatically collect statistics and reduce the data to useful summary form. Scheduling algorithms to control the flow of transactions in time are built in. Time is represented by a simulated clock which is automatically advanced to the time of the next event to be executed by a transaction. Time is assumed to pass in discrete steps; the smallest unit of time recognized by the simulator is chosen by the user.

example 1

Typical computer system areas where GPSS can be advantageously used are illustrated by three studies that used the GPSS language. First, consider a detailed study of equipment operation at the miscrosecond level. The case in point was a study of the interaction of several display terminals with a single display control unit. This control unit had to multiplex parallel character streams arriving from the display keyboards, which caused some delays in processing characters. These delays were occasionally long enough to be noticeable by a terminal operator. The object of the study was to find out how often this might be expected to occur.

The logical flow of characters through various gates of the equipment was modeled; circuit timings as small as ten microseconds were included. The intricate multiplex scan formed the heart of the model. The system logic was far too complex to be

represented by a queuing-theory model. Of course, much of the complexity may well have been irrelevant, but nobody could be sure of this until the system action was established (either by detailed simulation or direct experimentation with actual equipment). Because the action to be studied was mostly contained within a subsystem, it could be isolated from the larger computer complex of which it was a part; all interaction with the larger system was represented by simple delays. Because the extent of the model was greatly restricted, it was feasible to model with a very short unit of time.

example 2

Another study using GPSS considered the effectiveness of various queue-ordering schemes in reducing response time and main-storage requirements in a disk file configuration. Here, the concern was less with detailed timing than with detailed logic. The operation of the disk drives was easily modeled. A random source of disk accesses was generated to represent the demands of the rest of the system. In between, the queue-ordering logic permitted such schemes as picking the command with an addressed track number closest to the current one, or picking the shorter of two queues if duplicate files were available. Queuing theory presently does not give adequate models for such schemes. Moreover, there was no point in going into a full-scale system model. The limited scope of the model and the number of different schemes to be studied made GPSS the logical candidate for the study vehicle.

example 3

An example of a third kind of study, one leading to a rough overall view, was a model of an airline reservation system. Messages arrive over high-speed telecommunication lines and are received and queued by the interrupt-control circuitry of a central computer. The computer also provides scheduling for the disk-access request queue. Finally, message responses are sent back via the high-speed lines. The purpose of the study was to estimate total message transit time in the central computer and the amount of required main storage.

Everything was highly simplified. For example, channel interference was neglected and the effects of line polling ignored. The study gave a very rough indication of the operation of the system, i.e., whether certain files would be overloaded, or peaking conditions in main storage required attention, or the like. Equally important, the study provided designers with a better understanding of the system dynamics.

One caution concerning rough simulations of this nature. To avoid complexity, dependent events are assumed to be independent and branch points based on dynamic parameters are replaced by statistical branches, etc. If too much of this sort of thing is done, equivalent results can be obtained just as easily, more quickly, and more cheaply with analytic techniques based on queuing theory. On the other hand, if a great deal of complexity is essential, a more appropriate simulator should be considered.

The facilities and basic operations in a special-purpose simulator

special-purpose simulator

are likely to bear greater resemblance to the devices and operations in computer systems than in the case of GPSS. Thus, the language may refer to drums, disks, and terminals, and may well provide for an explicit program structure with commands such as READ, WRITE, SEEK, and BRANCH. This effectively restricts the application range to computer systems, but within this range, the simulator should be more natural to use than GPSS. In GPSS, the simplicity of the implicit flow from block to block is lost when a set of block routines is built to model specific computer operations. Some alternate flow mechanism has to be adapted to provide a program of operations.

In the beginning of a study, a special-purpose simulator of this nature may appear too complex, requiring more detail than is accurately known. For instance, a model for Examples 2 and 3 might require dummy operations with no significance. Since there was no need to carry on and amplify the models, the extra trouble in putting together such a gross-level model from such a specialized language might not be worth the effort. However, continuing development of a complex computer system model can tax the potentialities of GPSS rather severely. The typical study requires that a model be frequently updated to evaluate new schemes and modes of operation as they are proposed and formulated. A special-purpose language in which the principal entities represent devices, programs, messages, and commands can ease the problem of adding operational details to a model.

A carefully chosen modeling language not only permits one to simulate equipment operation but also provides access to simulator facilities such as queues, tables, variable statements, and functions. To the degree that these entities and operations are more specific and concise than their cognates in GPSS, a model can be more easily prepared and augmented. The price of this gain is that the simulator is likely to be very awkward for modeling other kinds of systems.

As in the GPSS case, the typical special-purpose simulator will simulate uncertainty and variability by means of random-number generators and will order anticipated actions on a "future events chain." Whenever the most imminent event is removed from this chain, an internal "clock" is updated. The event is then decoded and the specified action executed. The basic unit of time may be fixed, say at a fraction of a millisecond, to realize a compromise between the detailed microsecond level of internal machine actions and the millisecond level of external I/O actions. (Data on persistent phenomena, such as channel interference, can still be automatically accumulated on a microsecond basis.) Where microsecond time units, or on the other end of the scale, minute or hour units, are important to the study of a piece of equipment, GPSS should probably be considered.

Basic input in building any system evaluation model includes a statement of the system configuration with timing characteristics of the units in the system, a description of the job environment,

and a description of the system programs in suitable language. Statistical output from a special-purpose simulator can be of two types, the first being provided automatically, the second being user-requested. Where applicable, the former may include:

- Input specifications and model programs
- Device utilizations
- Statistics on queues
- Statistics on main storage requirements
- Average data rates per channel
- Time distributions for different types of messages to traverse various parts of the system

The user may also want to define outputs that include time averages for various processes, as well as counts of the occurrences of interesting events. The simulator design may also permit the user to intervene in the normal process and collect information not ordinarily generated by the simulator.

As suggested in the schematic of Figure 3, a study might typically model an on-line data collection system consisting of a SYSTEM/360 MODEL 30 with four 2311 disk storage units and four communication lines, each with several 1031 terminals. The equipment and the connections among units in such a configuration should be easy to describe in a well-planned modeling language.

In operation, a simulator can generate messages and simulate message transfers between the terminals and the CPU. Programs that the user has written in the modeling language can be used in simulating message operations, file accesses, main storage allocations, and CPU responses. Provision can and should be made for simulation of multiprogramming and I/O overlapping.

It is useful to distinguish among three kinds of instructions in the modeling language. Some of these instructions need to be related to system macroinstructions for reading, writing, branching, allocating, etc. Others need to be included for statistical sampling purposes only. Finally, some instructions are needed for manipulation of the simulation model; these may well include

Figure 3 Configurator schematic of a data collection system

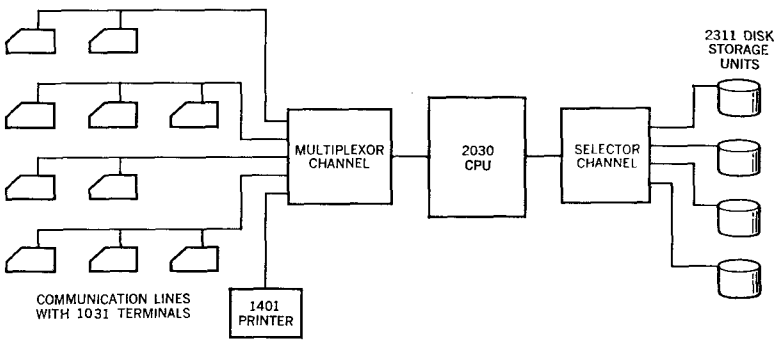
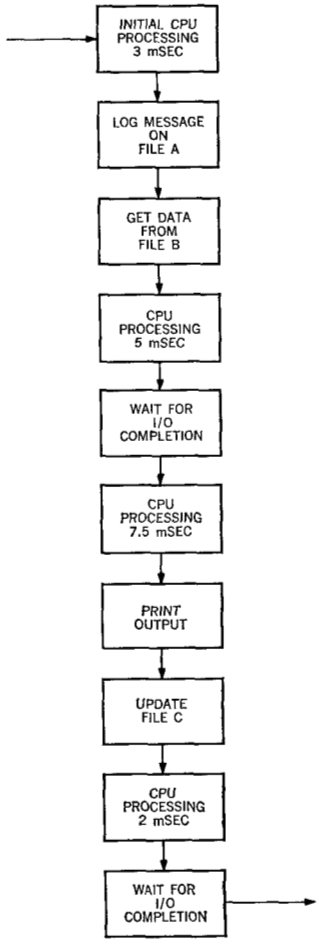


Figure 4 Flowchart of a typical operational program



a probabilistic branch, test instructions, arithmetic and queuing instructions, etc.

In modeling system programs, moreover, three classes of programs clearly need to be defined: application, control, and interrupt. The first of these represents application programs that perform the tasks the user wants done. The last two classes represent the operating system that controls the CPU and schedules the I/O requests.

A hypothetical application program that processes a particular type of message is flowcharted in Figure 4. This can be simulated by a sequence of the general form: process, write, read, process, wait, process, write, write, process, wait, and exit. Processing times can be simulated by assigning estimated elapsed times to the process instructions. The instructions for reading and writing require parametric specifications, such as arm number, track number, and byte count of the information being transferred. The user needs the flexibility of either stating a known arm, track, and byte count, or of making a dynamic choice by testing the state of the system or programs. In some cases (as in the initial phases of design), the user may also want the simulator to randomly select parameters from within specified ranges.

A wait instruction is assumed to suspend processing on the message in question until its outstanding I/O is completed. During this time, processing can begin (or continue) on other messages to simulate the multiprogramming environment. One way of handling this is to require that certain instructions be used to check lists or queues to ascertain whether other messages exist in the system and, if they do, where to begin processing them. In this way, provision can be made for entering and returning from other application programs at several levels and accounting for the necessary I/O operations whenever programs must be retrieved from auxiliary storage.

There are several such places where it is obvious that the simulator logic does not know by itself what to do next. All of these are analogous to the case in which actual monitor or control programs are called to decide the task a computer is to execute next. In a like manner, if the simulator automatically

control programs

Figure 5 Schematic of a control loop

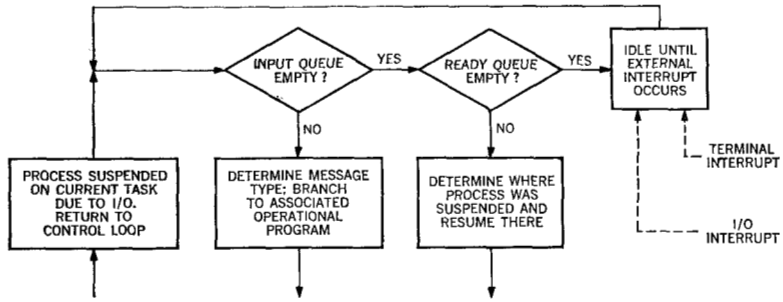
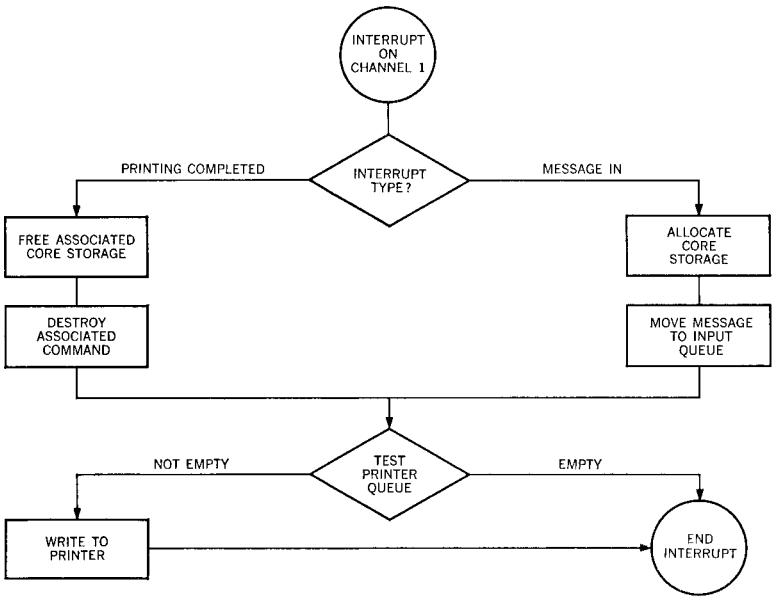


Figure 6 Schematic of a line-control program



branches to specific user-defined subroutines that determine what to do, the subroutines may simulate virtually any operating discipline. An example might be a task-scheduling loop, of the type shown in Figure 5, that is called whenever a wait instruction is encountered. A convenient modeling language for this loop needs instructions for testing the status of a queue, determining message type, and branching to relevant programs. The input queue is first tested to see if it contains any new messages. If so, the first message is removed and made the current message. Assuming that a stored table contains a suitable list of program addresses, the proper operational program for a message of the current type is ascertained and entered by a branch. But if the input queue is empty, a branch is made to test the ready queue (a list of partially processed messages ready to go again after having been suspended because of incompleted I/O operations). If messages are found in the ready queue, one of them is made the current message and its processing restarted. If both queues are empty, the CPU enters an idle status.

interrupt programs

The third class of programs that needs to be distinguished in modeling consists of interrupt programs. These can be entered, interrupting any application program in process, whenever a change in status occurs in either an I/O operation or an interval timer. Thus, an interrupt may be simulated when a data transfer has been completed, or a disk arm completes a seek operation, or a time period has elapsed. For example, shown in Figure 6 is the logic of an interrupt program that processes communication-line interrupts when messages enter the simulated computer system or data is printed out. A modeling language for easy

simulation of this program needs instructions for branching on interrupt type, allocating core, releasing core, message queuing, testing queue status, and message transmission. Initially, a test determines whether the interrupt is caused by a message entering or data leaving the system. If a message is entering, main storage is allocated and the message placed on an input queue. If data has just been printed, main storage is freed, the I/O command controlling the print operation is destroyed, and the printer queue is then tested. Printing is initiated if a message is found. Finally, the interrupt is terminated and processing reverts to normal mode.

In a special-purpose simulator, it is entirely possible for model programs to exist as sequences of instructions similar to the programs of any symbolic coding language. Instructions may be executed interpretively and addressed by a pseudo-counter that is automatically updated. If the instructions in the modeling language are primitive operations for the simulator, they need not be fashioned from other operations, as is required in GPSS. Devices may be defined at the start of the simulation run, and various hardware parameters, such as seek characteristics and rotation time, may or may not be built into the simulator.

The major requirement for a simulator user is to specify how special activities, such as interrupt control in a channel program, are to be carried out. In a simple case, it may appear that these activities pose unnecessary complications and that the channel control should be made automatic. However, a special-purpose simulator becomes most useful in complex cases where such routines are left to the user's specification. When control routines becomes standard, they are usually understood and there is little point in simulating them; more often than not, it is the unusual that is simulated. The flexibility made possible by divorcing control operations from predetermined hardware responses can permit a suitable special-purpose simulator to model situations that would be very difficult to model in GPSS.

Considerations in validating simulation results

A pertinent but hard-to-answer question is, "How does one know whether a simulated result is correct?" The question requires consideration of two independent avenues of inquiry, one relating to the accuracy of the model, and the other to the precision of the statistical results.

A model is accurate to the extent that a valid abstraction has been made from the proposed system. A high degree of accuracy requires that the many simplified or omitted details do not markedly affect the aspects of system performance under study. One can never be completely sure of an experimental design, but some of the more obvious assumptions can often be checked. For instance, in one study it was suggested that mean message length was significant, but that variation from the mean was not. Tests comparing the results obtained with constant message lengths against those obtained with variable lengths uniformly

spread about the mean validated the assumption for the range of interest.

In some cases, details with a decided effect on system performance may not be firmly known at the time of a study. As a consequence, the model cannot be highly accurate, and estimators obtained from the model cannot be taken as predictors of absolute performance. But much can be done with such a model in a relative sense. By comparing runs with and without a change, the relative effect of the change can be measured. In making relative comparisons, however, one must try to consider the relationships between known and unknown variables. For example, for one model of a conversational mode system, it was expedient to generate random inputs from terminals rather than try to reconstruct actual conversations. This was satisfactory as long as the results were limited to the processing unit and file logic. However, since the message-generation process can influence the effect of a particular polling discipline, a study of various polling schemes would have been inappropriate with such a model.

In constructing a model, it is well to build from the known to the unknown. For instance, a model might be exercised with random input and the results checked against known performance curves. Then, if a queue-sorting technique is introduced, there is not only a basis for comparison but some assurance that any derivations from the base are due to the new technique and not to an error in model logic. This is important; in complex models, there is ample opportunity for logical errors. In simple simulators, errors in logic usually reveal themselves in absurd statistics; in larger simulations, however, error may lurk in the most reasonable looking results. For this reason, tracing techniques should be abundantly employed. If several message types pass through the simulated system, each type should be entered separately to verify that it follows the specified path. If probabilistic branches are employed, checks should be made to ascertain that paths are used as anticipated. At the end of a run, all residual messages and counts left in the system should be explainable. Minor inconsistencies may be the only trace of a blunder other than a legacy of invalid, though apparently correct, statistics.

Unexpected results should be thoroughly investigated; these are often due to improper sampling or logical error. If results are to be believed, they must be explicable as well as repeatable. A simulation study is incomplete unless the output is supported by a thoughtful rationalization for all observed behavior.

The second line of inquiry into the correctness of simulation results relates to statistical validation. This problem arises from the nature of the sample process inherent in the technique of simulation rather than from the accuracy of the model chosen to represent the system.

It is assumed that any statistic of interest in a model, say response time for messages of type A, has a particular distribution with a true mean value. Simulation estimates these statistics by

may appear overloaded. In such a case, sample size should be measured, not by the number of ordinary messages processed, but by the number of these infrequent messages processed. A more useful picture of system operation describes two system states (one without the infrequent occurrence, one during the occurrence) with a statement of the frequency of such occurrences and the length of the transition between the two states. An example of this sort of problem occurs in the study of "graceful degradation" of a system. The model may be based on system events, such as processing times, seeks, and responses, measured in milliseconds. The failure events causing the system to be degraded from one state to another may be measured in tenths of hours. The only reasonable thing to do is to simulate system operation separately in various degraded states and then combine these results in a sort of system profile.

Finally, it is well to emphasize that simulation studies are experiments run on a system model in lieu of the real system. Thus, the principles that apply to other experiments apply. These principles need not be discussed here; they are treated in numerous texts on the design of experiments. These principles are apt to be neglected unless the simulation study is viewed as an important investment deserving of careful planning.

Summary

This paper attempts to point out the role that simulation plays in the design of computer systems, emphasizing that this technique should be used, not as a sporadic tool to obtain unrelated answers, but as an experimental method and a continuing integral part of the design process. Although simulation is especially pertinent to the design of teleprocessing systems, it should be emphasized that simulation methods can also be useful in the design of the broad spectrum of computer systems including time-sharing and multiprocessing systems.

CITED REFERENCES AND FOOTNOTES

1. B. Dimsdale and H. M. Markowitz, "A description of the SIMSCRIPT language," *IBM Systems Journal* **3**, No. 1, 57–67 (1964).
2. R. E. Efron and G. Gordon, "A general purpose digital simulator and examples of its application," *IBM Systems Journal* **3**, No. 1, 22–34 (1964).
3. H. Herscovitch and T. H. Schneider, "GPSS III—an expanded general purpose simulator," *IBM Systems Journal* **4**, No. 3, 174–183 (1965).
4. References to other simulation languages may be found in the bibliography of a recent simulation article: D. Teichroew and J. F. Lubin, "Computer simulation—discussion of the technique and comparison of languages," *Communications of the ACM* **9**, No. 10, 723–741 (October 1966).