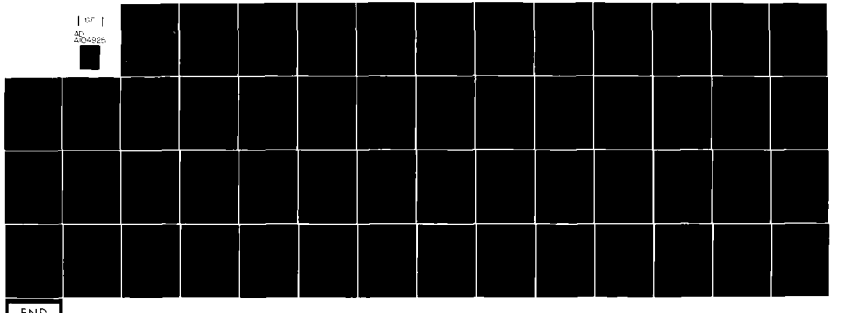
AD-A104 925 BATTELLE COLUMBUS LABS OH F/G 9/2
COMPUTER SYSTEMS SIMULATION: AN OVERVIEW.(U)
JAN 80 L L ROSE

UNCLASSIFIED NL



END
DATE FILMED
10-81
DTIC

15

FINAL REPORT

on

COMPUTER SYSTEMS SIMULATION: AN OVERVIEW

to

UNITED STATES ARMY INSTITUTE FOR RESEARCH IN MANAGEMENT INFORMATION AND COMPUTER SCIENCES

January [redacted], 1980

by

Lawrence L. Rose

BATTELLE
Columbus Laboratories
505 King Avenue
Columbus, Ohio 43201

The views, opinions, and/or findings contained in this report are those of the author and should not be construed as an official Department of the Army position, policy, or decision, unless designated by other documentation.

This research has been performed for AIRMICS through the United States Army Research Office under Contract D.O. 1372.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

LIST OF FIGURES

ABSTRACT

This report addresses the appropriateness of simulation languages as auxiliary tools to help the USACSC attain its objectives. The basic concepts of simulation are overviewed; discrete event languages are shown to be consonant with USACSC modeling requirements. The three major discrete event languages in use today (GPSS, SIMSCRIPT, and GASP) are overviewed. Selection criteria are derived and employed to assess the use of these general simulation languages in the USACSC environment. The conclusion is that languages more computer systems oriented than these general simulation languages are preferred for USACSC utilization. Although time precluded an additional intensive evaluation of these specialized languages in this study, previous research in this area was noted. Based upon this background, the languages IPSS and ECSS are recommended as representative of the tools most appropriate for USACSC simulation uses.

1. INTRODUCTION TO SIMULATION

To simulate an activity is to mimic its behavior: given the same inputs as the real activity, the simulation model creates outputs that appear indistinguishable from those of the real activity. More explicitly, author R. E. Shannon [1] defines simulation as follows:

> "Simulation is the process of designing a model of a real system and conducting experiments with this model for the purpose either of understanding the behavior of the system or of evaluating various strategies (within the limits imposed by a criteria) for the operation of the system."

Computers, both analog and digital, have been used heavily to support simulation for some time now. Languages, with particular orientations to modeling analysis, have been developed to ease the modeler's work and to increase the degree of complexity of the simulation.

The objective of this chapter is to familiarize the reader with the basic concepts of simulation and to place the computer's role in modeling into the proper perspective. Not only must modelers be cognizant of the essential elements of simulation, so also must the decision-makers who base decisions on model outputs. Without some knowledge of modeling, simulation results are difficult to sell and apply; it is the modeler's job to carry out the simulation in a meaningful, documented manner so that management can be convinced to utilize the simulator.

1.1 Why Simulate?

The motivation for simulation stems from man's need to learn something more about the "system" under consideration. The modeling process is beneficial in and of itself, as we cannot construct a suitable simulator unless we can adequately define the subject "system". This often results in a clear, concise problem definition, which in some cases may be the most difficult part of the solution.

If we already understand a system, then the modeling activity may be desired to avert the cost, technology, morality, time, risk, or applicability factors. The presence of any of these factors is normally sufficient to warrant simulation:

- Cost : A prototype may be too costly to build; often we are selecting from a large set of design alternatives, all of which cannot feasibly be built given the cost constraints. Possibly the construction cost may be feasible but testing costs too high to carry out.

- Technology : The prototype may be impossible to construct using today's technology but will be available in the future so we wish to model the design today to better prepare for tomorrow.

- Morality : Certain models cannot be executed "real-life" because of moral problems, e.e., pilot crash-landing procedures, simulated war, etc.

- Risk : Building and bridge stress designs, for example, are much less risky to simulate than to prototype. E.g., placing bumper-to-bumper trucks on a bridge with 80 wph winds to test stress capability carries no risk to the bridge under simulation.

- Applicability: Often simulation is performed as a last resort: mathematical models are insufficient or undefined; prototypes cannot be constructed without further knowledge about the activity. Construction of a simulation model may lead the way to design/understanding breakthroughs.

Given a completed simulation model, we have overcome the cost, technology, morality, time, or risk factors which prohibited construction of an actual test system in the first place. With the model, and suitable

"hooks" in the framework, one can exercise the simulated system to answer what-if questions that otherwise are only conjecture. The role of statistics becomes more vital as the desired precision of the results increases.

Effective simulation, then, is a combination of accurate modeling followed by valid testing procedures. Most simulations are not trivial exercises; they are time-consuming, labor-intensive, and expensive. The potential gain lies in the ability of the simulator to affect decisions of some larger magnitude in a positive manner.

1.2 Simulation Languages

Computer languages provide appropriate support for simulation activities that involve either complexity, magnitude, or repetition. Figure 1-1 illustrates a structured delineation of the most popular of these simulation languages.

General purpose languages, such as FORTRAN, COBOL, and P1/1 continue to be used effectively for simulation. However, they are not oriented to simulation. They were designed for scientific, business, and multi-purpose processing, respectively. These languages have no explicit facility to model systems changing over time, for instance, or queues, or single/multi-server units. Further, no statistics gathering statements are found in these general purpose languages. Hence, successful simulation with one of these languages requires a sophisticated modeler, highly knowledgeable in both simulation and the host language. The resultant model is normally highly inflexible, and opaque to the user as the code does not clearly reflect the concepts modeled. Changes in statistical requirements are normally difficult to accommodate, thus making the model highly specialized and costly.

Steps were taken to design and implement simulation-oriented languages mainly due to the inappropriateness of the existing general purpose languages. The tools were too clumsy for the designer to use effectively and efficiently. The result was a potpourri of languages for simulation, all aimed at closing the gap between design concept and implementation, with suitable enhancements for testing and analysis.

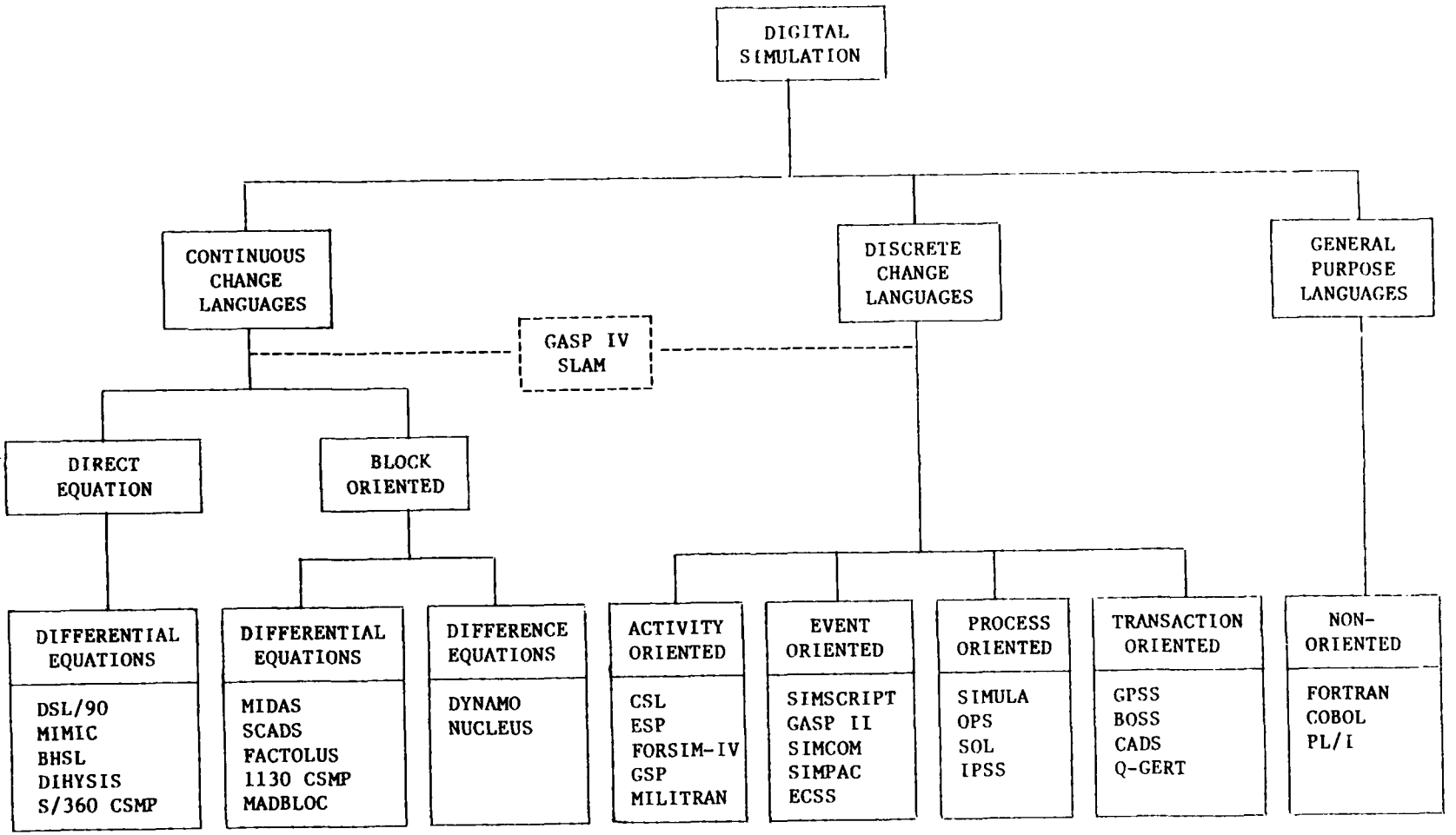


Figure 1-1. Simulation Programming Languages [1]

Simulation languages are classified into two major categories: continuous and discrete. Continuous simulation languages focus upon aggregate items in infinitesimal time increments, whereas discrete simulation languages focus upon individual items in ad hoc time increments in accord with the next imminent event. Thus, continuous simulation languages are utilized when one can describe the system as a continuous process, i.e., there exist functional definitions of the process with time as a parameter.

Continuous processes are normally described by a series of difference equations or differential equations. Hence, continuous simulation languages such as DIHSYS, DYNAMO, and NUCLEUS have been developed to host the modeler's functional definition of the system to be modeled. Centered upon a set of difference or differential equations are capabilities to initialize, start, reset, etc. along with summary and statistical output features. The crux of any continuous simulation is the functional definition of the "system":

$$\left\{f(x_1,x_2,\ldots x_n) = (y_1,y_2,\ldots y_m)\right\} \quad .$$

Continuous simulation relies upon the veracity of each internal equation, and the assumptions on which each holds. The main deterrents to continuous simulation are twofold:

(1) The process is so complicated or vague that the set of characteristic equations cannot or is not defined; and/or

(2) The assumptions required for the underlying equations cannot be realistically made for the simulated system.

Discrete simulation languages have been under development for over 15 years, and in the last five years they have proved extremely valuable to the modeler. They provide constructs such as a time clock, next events queue, data structures, statistics, dynamic transactions, etc. automatically to the modeler. Thus, one can, when utilizing one of these languages, concentrate on the definition of the model as opposed to the

implementation of the model on a computer in some alien language. One defines a discrete simulation model by detailing all of the events/acts pertinent to the system and the actions which trigger them. The model is defined at a macro or micro level depending on the detail of the design or the requisite precision of the output.

Four types of discrete simulation languages can be distinguished:

(1) Activity-oriented,

(2) Event-oriented,

(3) Process-oriented, and

(4) Transaction-oriented.

Shannon [2] provides the following definitions, which we amplify.

Activity-oriented languages represent time-dependent activities as instantaneous occurrences in simulated time. Thus, one does not schedule occurrences, rather one specifies under what conditions they can happen. The program is composed of two major sections: a test section to determine what activities can now occur, and an action section to update state and time conditions.

Event-oriented languages represent an event as an instantaneous occurrence in simulated time, automatically scheduled to occur when it is known by the model definition that the proper conditions exist for its occurrence. All events and their interactions are defined independently; an executive program can automatically sequence all scheduled events.

Process-oriented languages are a hybrid derived from the concise notation of activity-oriented languages and the efficiencies of event-oriented languages. A process is a set of events; it is dynamic and can exist over time. Processes can be interrupted, have sub-processes, and can be reactivated. The executive program controlling these processes is necessarily more complex than that required for event/activity-oriented languages.

Transaction flow-oriented languages use the block structure of flow charts to describe the simulation, with transactions flowing through activity and time altering blocks. Each block specifies specific actions, with restrictions on parallel execution, etc. These languages are extremely easy to use but much less flexible that the other types.

1.3 Discrete Event Simulation

Regardless of whether the discrete simulation language is activity, event, process, or transaction-oriented, the essence of discrete event simulation is shown in Figure 1-2 (see Mihram [2]). Central to simulation is the notion of a clock, which governs the time parameter t. The initialization portion of the simulator must be able to characterize the system's initial conditions, to include start time, initial system entities (objects) and attributes (properties). For example, a bank opens at 9 a.m. with three clerks named Bob, Sue, and Rose, and two customers at the front door to deposit $150 in cash and withdraw $20, respectively.

Data structures appropriate to simulation, such as queues, stacks, and sets must somehow be provided the modeler, either explicitly or implicitly. These structures must have suitable hooks so that statistics regarding their use, length, etc. can be gathered. Reporting facilities are a requisite capability so that the modeler can create his desired outputs for analysis and conclusions.

Execution-time controls are necessary so that the model, once deemed correct by the builder, can be exercised to provide meaningful test series which lead to unbiased outputs. In this way only can the model be verified and meaningful results obtained.

Three of the major discrete-event languages used today are GPSS, SIMSCRIPT, and GASP. These languages offer statement constructs to alter the time clock, gather statistics, produce output reports, initialize and end simulation studies, automatically handle queues for the modeler and automatically control event selection.

On the other hand, these languages are worlds apart in their basic methodology. The statements are all different and the model construction almost completely dissimilar, yet these languages can be used to model transportation systems, banks, airports, etc. Thus, it is felt that a short exposure to each language would be beneficial to the reader; all sides of this language triangle have intrinsic value.

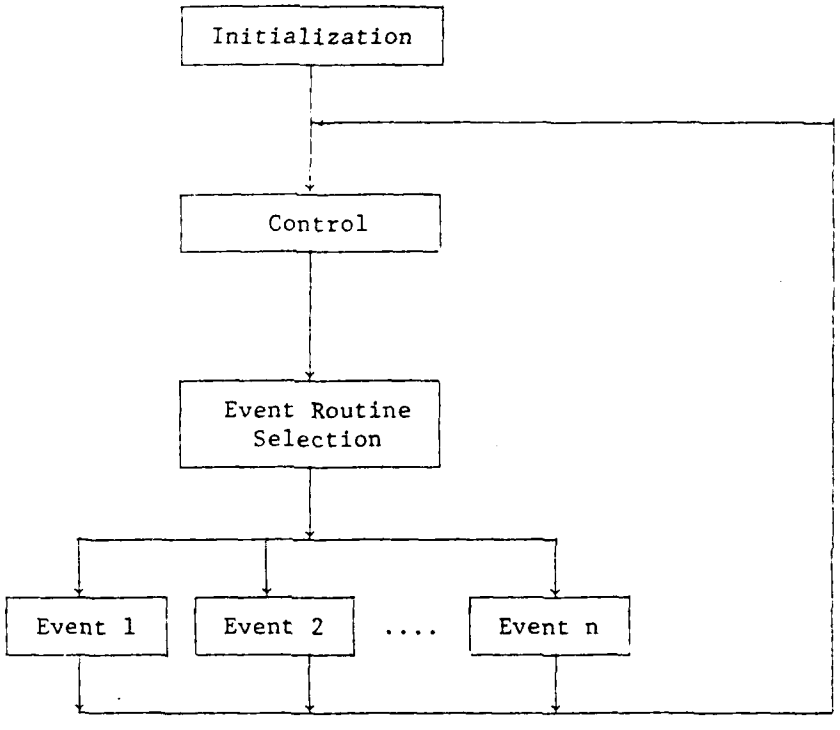


Figure 1-2. The Simulation Algorithm [2]

Components:

- clock
- data structures
- statistics-gathering
- report generation
- process description
- execution-time controls

It should be noted that neither GPSS, SIMSCRIPT, or GASP has constructs especially oriented towards describing computer systems. Nonetheless, many simulations of computer systems (especially at the macro level) have been carried out using these languages. Chapter 5 will further address this issue. First, let us proceed to examine the GPSS, SIMSCRIPT, and GASP worlds of discrete event simulation. To provide a thread of continuity between the discussion and comparison of these three languages, let us consider a simple, but computer-oriented example that can then be modeled in each language.

The example should be simple enough to keep the simulation models concise, yet rich enough to enable a demonstration of the subject languages. Outputs of some import must be available so that statistics collection can be demonstrated and results compared. Lastly, the example should be in the computer systems domain as that is our topic area of interest.

The objective of actually generating models of an example system is to show not only the capabilities of the subject languages, but also the resources required (both time and knowledge) and resultant model complexity. This should also provide the reader with a reasonable level of understanding of these three languages and their role in modeling.

2. A SIMULATION EXAMPLE

The chosen example is a very high-level model of a computer system. Only at the macro level can concise models be constructed. While this example appears to be very simplisitc, it is representative of an entire class of extremely valuable and useful queueing models. Hence, we are laying the groundwork for a model which can be further refined into particular systems configurations.

Figure 2-1 shows the configuration of our macro model. Jobs come in, use the cpu, and output results. No utilization of other hardware or software devices is considered. (Actually, this is what one considers if cpu utilization is the problem and is independent of other factors such as disk seek time.)

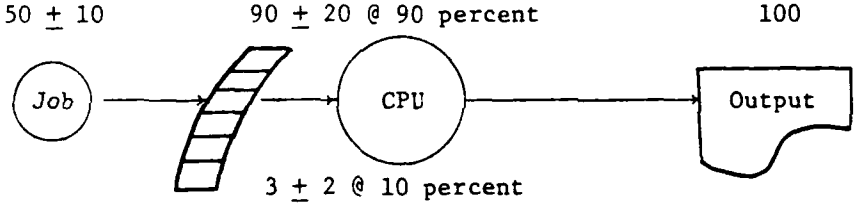


Figure 2-1. Single Server Cpu Example

No multi-programming is assumed, although that would be a simple model extension. Hence, a job must enqueue if the cpu is busy. No wait for output is assumed, so it is not shown in our models. Several parameters are required to enable model definition and execution: inter-arrival time, service time, and simulation duration.

Inter-arrival time is used to determine the time of the next arrival. Since it represents the time between arrivals, one can compute the time of arrival i+1 to be the time of arrival i, plus the inter-arrival time. Almost all simulators utilize this to alleviate the problem of storing many future arrivals needlessly. Our example assumes that a job arrives every 50 seconds, $\pm$ 10 seconds. The distributions of this variance and that of cpu service time are assumed to be uniform.

Service time is the time required to perform some service. In our case it represents job execution time. Our model will show the cpu as a resource with a service time of 90 $\pm$ 20 seconds for the 90 percent of the jobs that run to completion, and 1 to 5 seconds for those that abort.

Simulation duration specifies the stopping criteria for the model. It is typically expressed in time units (simulate a day, a week, a quarter, 27 years) or transactions (build 1,000 cars, make 593 babies) or events (when the bin is empty, when the bank is broke). In this case we have chosen the simulation to go on until 100 jobs are completed.

The next three chapters will discuss and illustrate the GPSS, SIMSCRIPT, and GASP languages and how they can be used to characterize our example model.

3. GPSS

The General Purpose Simulation System (GPSS) is a transaction-oriented simulation language developed by IBM [3]. As such, it is widely available and well supported by IBM. GPSS was initially developed in 1961, and has undergone several upgrades.

GPSS is an easy language for the user to mold models. A flow diagram with special symbols for the basic GPSS constructs transforms directly into GPSS code. GPSS automatically handles the synchronization of events; future events are triggered automatically given the user-defined inter-arrival times. Output statistics are automatically generated and formatted for the user; literally no programmed I/O is permitted the GPSS user.

3.1 Language Constructs

GPSS basic constructs include automatic maintenance of queues (priorities are possible also), GENERATE to set up arrivals given inter-arrival rates, QUEUE/DEPART commands to generate queueing statistics, FACILITIES to handle a specified number of elements concurrently, an ADVANCE command to increment the clock and hold a facility, MARK and TABULATE commands for output accumulation and event time statistics, TRANSFER for probabilistic movement through the flowchart, and TERMINATE to end a given transaction in the system.

Typical GPSS modeling examples include: ships in a harbor, manufacturing shops, hair salons, gasoline stations, inspection shops, and simple queuing server models. Figure 3-1 details the essential GPSS capabilities in block form. Twenty statements are available to the modeler to describe the process, with another ten statements to govern the simulation execution.

Transactions are created at each GENERATE block, and travel through the network as defined by the modeler: WAITing for storages/facilities, consuming time as ASSIGNed, and being GATEd to the next appropriate block. Twelve parameters are available (by default) to the GPSS modeler to hold values with every transaction.

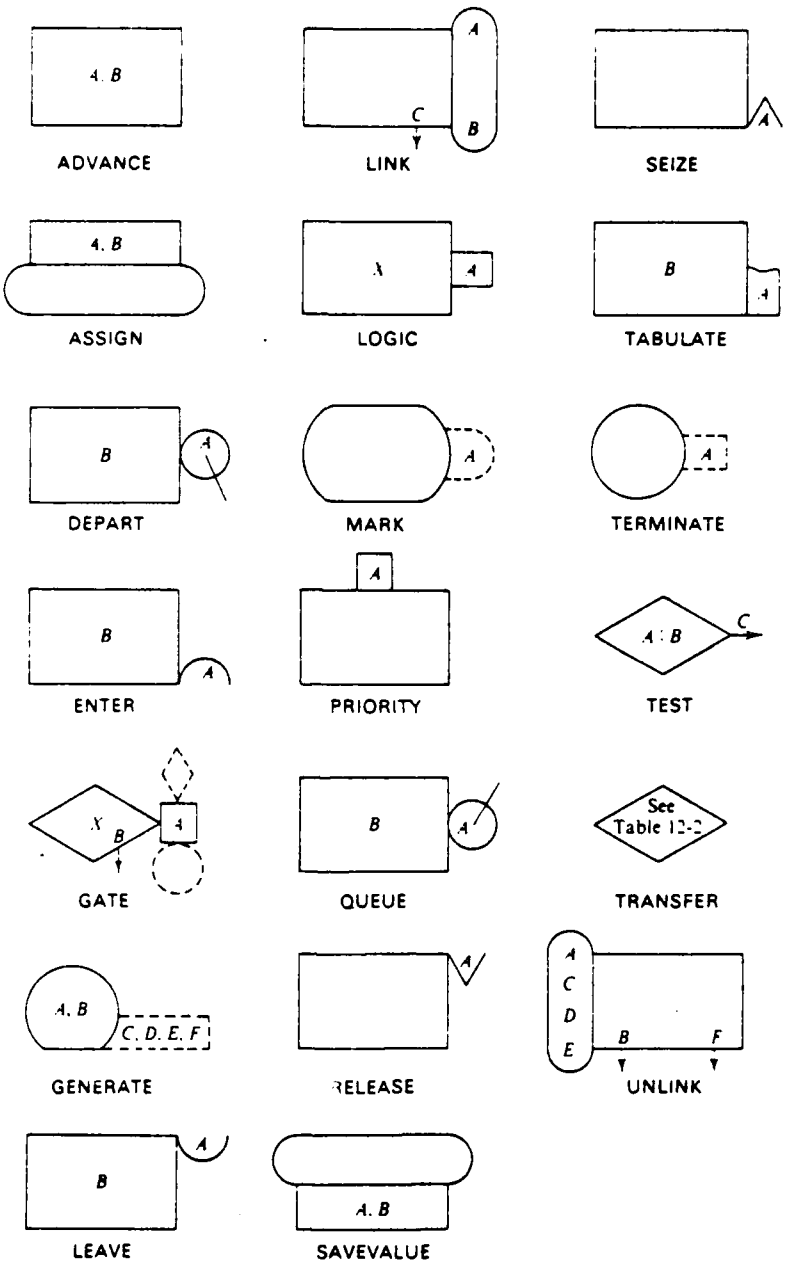


Figure 3-1. GPSS Block-Diagram Symbols [4]

The great strength of GPSS lies in its simplicity and one-to-one correspondence with a GPSS flow chart. This is by far the preferred language for neophyte users or anyone requiring a simulation of a process that is not overly difficult to describe. The program mirrors the description and thus appears easy to verify. Output statistics are automatic; almost every GPSS model runs and provides output. The system is moderate in size and cost, but is generally confined to IBM hardware.

GPSS provides much to the user to ease the modeling task. The payment for this is great inflexibility. Only an expert can weave GPSS code with Assembler subroutines to add features not resident within GPSS. GPSS provides a macro capability but not subroutines. Thus, GPSS programs of any complexity become terribly long and cumbersome to alter, modify or debug. There are no particular constructs in GPSS to model either computer systems or data base systems.

3.2 Single Server Model

The essence of GPSS is best shown by a short example. Consider now our high-level model of a computer system where jobs arrive every 50 ± 10 seconds and execute serially for 90 ± 20 second except the 10 percent that abort within the first 5 seconds. To generate cpu utilization for 100 jobs, we write the GPSS flowchart and associated code as illustrated in Figures 3-2 and 3-3.

Given that we are explicitly modeling the queue for the cpu (GPSS will characterize it, even if we choose to ignore it), the QUEUE and DEPART statements will enable GPSS to derive queue statistics for us. Note how closely this GPSS flowchart mirrors the modeled system (Figure 2-1).

Clearly this GPSS example could just as easily model incoming phone calls, some of which are wrong numbers (abort), the rest of which we talk awhile and then hang up. Implicit in the SEIZE statement is an invisible queue, in which transactions (representing jobs in our case) automatically line up if the facility (representing the cpu in our case)

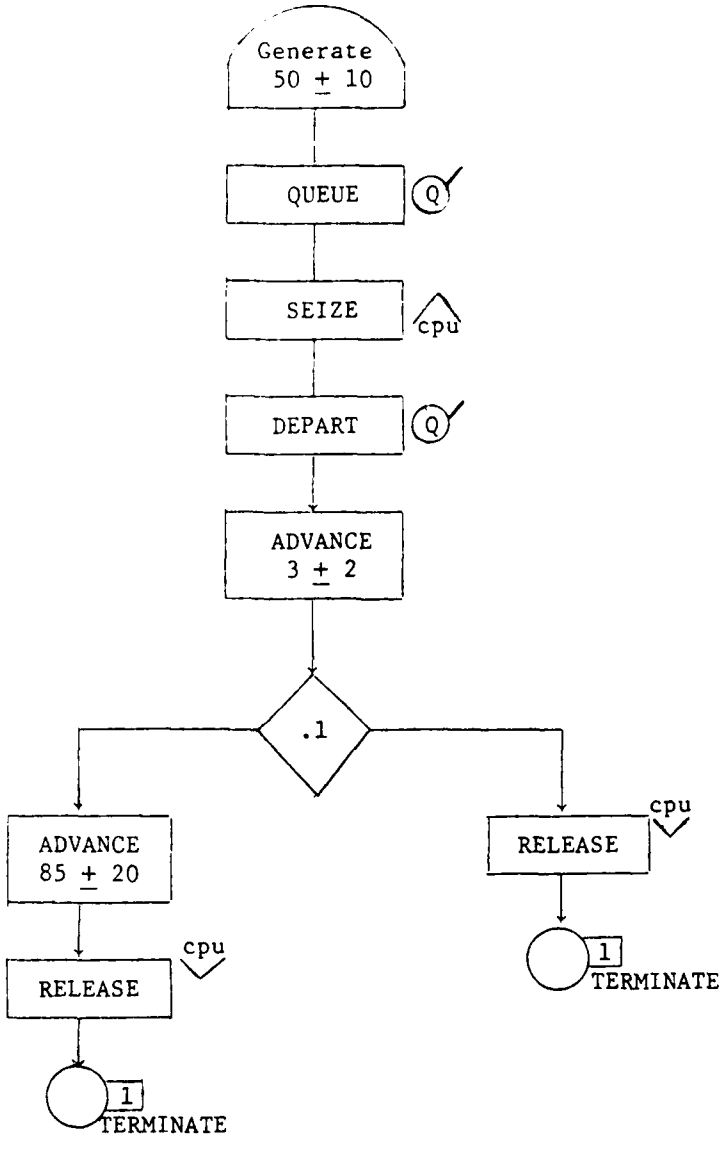


Figure 3-2. GPSS Example: Facility Utilization

```
* * *  G P S S / 3 6 0 / O S  V E R S I O N  2  * * *
*** IBM PROGRAM PRODUCT 5734-XS1 (V2M2) ***
```

```
                                                    COMMENTS
        SIMULATE
*
*
*       ..............................
*       .  GPSS SINGLE CPU EXAMPLE   .
*       ..............................
*
        GENERATE    50,10           ...JOB ARRIVES EVERY 40 TO 60 SECONDS
*
        QUEUE       CPUNE
        SEIZE       CPU             ...TAKE CPU WHEN AVAILABLE
        DEPART      CPUNE
*
        ADVANCE     3,2             ...RUN FOR 1 TO 5 SECONDS
        TRANSFER    .1,RUN,ABORT    ...10% NOW ABORT
*
RUN     ADVANCE     85,20           ...90% RUN TO COMPLETION...
        RELEASE     CPU             ...(FREE NOW FOR NEXT JOB)...
        TERMINATE   1               ...SUCCESSFUL JOB COUNT...
*
ABORT   RELEASE     CPU             ...(FREE NOW FOR NEXT JOB)...
        TERMINATE   1               ...ABORTED JOBS COUNT...
*
        START       100             ...HALT AFTER 100 JOB COMPLETIONS
*
        END
```

Figure 3-3. GPSS Example Code

is busy. Implicit in the RELEASE statement is the checking of transactions awaiting the associated SEIZE block (our singe cpu) and giving the transaction at the front of the line the desired facility. The clock is updated automatically and transactions pushed, in parallel or whatever our structure, through the blocks.

Hence, this GPSS language enables us to characterize asynchronous processing even though our definition is sequential; it is transaction oriented rather than procedure oriented as are FORTRAN, et al. The flowchart of Figure 3-2 shows the two paths each job may follow: either job abort after 1-5 seconds cpu or job completion after 66-110 seconds cpu. With wait time for the cpu included, final transaction time in the system could be considerably more. As we covered the SEIZE cpu statement with the sequence

```
QUEUE     CPLNE
SEIZE     CPU
DEPART    CPLNE
```

then the resultant GPSS output will provide statistics on this invisible queue CPLNE maintained for the CPU Facility to include maximum queue length, average queue length, etc. The TABULATE and TABLE commands can be added to provide further histogram-type data and transaction activity in any subpart of the modeled system.

3.3 Execution Output

The GPSS language provides a wide assortment of instructions to TALLY values, create TABLEs or histograms, and analyze queue, facility/storage usage. Illustrated in Figure 3-4 is the output resulting from our sample GPSS model of a single-server cpu. The initial portion of the output summarizes the system state at the end of the simulation. We see that 7,909 time units were consumed when the 100th job completed. Our 11 statement program is directly associated with block counts 1 to 11 (GPSS is interpretive). As block 2 is our queue CPLNE, we see that 56 jobs await execution. Block 1 total shows that 156 jobs were created, and of the 100 completed jobs, 10 aborted (blocks 10, 11).

RELATIVE CLOCK 7909 ABSOLUTE CLOCK 7909

BLOCK COUNTS BLOCK	CURRENT	TOTAL	BLOCK	CURRENT	TOTAL	BLOCK CURRENT
1	0	156	11	0	10	
2	56	156				
3	0	100				
4	0	100				
5	0	100				
6	0	100				
7	0	90				
8	0	90				
9	0	90				
10	0	10				
11	0	10				

FACILITY	AVERAGE UTILIZATION	NUMBER ENTRIES	AVERAGE TIME/TRAN
CPU	.993	100	78.559

QUEUE	MAXIMUM CONTENTS	AVERAGE CONTENTS	TOTAL ENTRIES	ZERO ENTRIES	PERCENT ZEROS
CQUE	56	27.321	156	1	.6

$AVERAGE TIME/TRANS = AVERAGE TIME/TRANS EXCLUDING ZERO ENTRIES

AVERAGE TIME/TRANS	$AVERAGE TIME/TRANS	TABLE NUMBER	CURRENT CONTENTS
1385.172	1394.109		56

Figure 3-4. GPSS Example Results

The single-server facility in our model was named CPU. We see utilization near 100 percent, as we suspected, with 100 jobs using CPU for a mean time of 78.6. Lastly, the queue statistics for CPLNE show a growing queue which was observed empty only once, and most jobs spent around 1,400 time units awaiting the cpu.

This demonstrates the power of GPSS: with a very few, natural constructs one can quickly and easily model systems of low complexity. Output capabilities are significant and facile to provide. On the other hand, GPSS models are difficult to debug--they very quickly represent complex branching and asynchronous processing. Lastly, GPSS models grow linearly: the model constructs do not help one program in a structured or modular or hierarchial manner.

Given this brief overview and example of a GPSS model, let us now move on to the other major languages: SIMSCRIPT and GASP. SIMSCRIPT is a true language (in the sense of compilation), whereas GPSS is interpretive and GASP is Fortran-based.

4. SIMSCRIPT

The SIMSCRIPT language [5] (taken from the combination of the words simulation and Sanskrit: simulation language) was developed by the RAND Corporation in the early 1960's and is currently maintained by CACI, Incorporated of California. Initially, SIMSCRIPT statements were transformed into FORTRAN which was subsequently compiled. Presently, SIMSCRIPT statements are translated directly into Assembler and versions exist for Honeywell, IBM, and CDC computers.

4.1 Language Constructs

SIMSCRIPT is an event-oriented simulation language. The modeler defines all events and event processors and the system controls the invocation and sequencing of events. The conceptual components of SIMSCRIPT are threefold; entities, sets, and event routines.

Entities (people, machines, jobs, etc.) have associated attributes (hair color, weight, cost, time, etc.) and can be either permanent or temporary in nature. Permanent entities (computer hardware, lumber machines) remain fixed throughout the simulation, whereas temporary entities (user jobs, clients) come and go during the simulation--they represent the dynamism in the model.

Sets are user-defined groupings (static or dynamic) of related entities. They may be ordered (waiting line or queue) or non-ordered (the set of disks) depending upon use. Statistics can be collected concerning set size, etc., during the simulation.

Event routines are used by the modeler to define the exogenous and endogenous events of the system under consideration. Exogenous events characterize external events (job arrivals) that cannot be controlled internally. Endogenous events are these events (job start and completion) that occur from within and trigger other events.

The power of SIMSCRIPT lies in the user ability to define these events independently, even though they may be inextricably interwoven. Each event routine is a subroutine, so changes in event characterization, etc., can be made without altering other aspects of the simulation.

SIMSCRIPT can be used to model the same types of situations as does GPSS. In addition, SIMSCRIPT provides more modularity and flexibility plus further numeric evaluation capabilities. This language is more sophisticated than is GPSS--the modeler must have a good programming background. Figure 4-1 lists the basic statements of the SIMSCRIPT language, which the modeler uses to construct his model.

Note that SIMSCRIPT provides normal procedure-oriented statements (FOR, WITH, DO, IF, GO TO, LET) in addition to simulation-oriented statements (CREATE, DESTROY, CAUSE, FILE, REMOVE, EVENT, SCHEDULE). This gives the modeler a rich set of language constructs to support statistical generation and testing during or in addition to the actual modeling. The idea is that this one language be utilized to support all aspects of the simulation activity, to include data collection, modeling, and output analysis.

4.2 *Single Server Model*

We contrast SIMSCRIPT to GPSS by generating the model as defined in Chapter 2. Although appearing considerably more complex, the SIMSCRIPT model has the advantage of being further tailored to almost any desired degree of detail, whereas the GPSS model is much more limited in scope.

The SIMSCRIPT view of this high-level model of running 100 jobs through a computer with some aborting before completion is to define two major events: job arrival (exogenous event) and job completion (endogenous event). Figure 4-2a illustrates the SIMSCRIPT preamble and main program, both user-defined.

The PREAMBLE sets up the supporting data structures, the statistics gathering procedures, and the event routines to be later defined by the modeler. In this case, three events will be synchronized (by time) to dictate execution of this simulation: ARRIVAL, COMPLETION, and STOP. Our simulation is viewed as being composed of job arrivals, subsequent job completions (or aborts) and finally a STOP criterion of 100 jobs. The only temporary entities are JOBs, each having three

CREATE "temporary entity or event notice" CALLED "variable"
DESTROY "temporary entity or event notice" CALLED "variable"
CAUSE "event notice" CALLED "variable" AT "floating-point time expression"
FILE "variable" IN "set"
REMOVE FIRST "variable" FROM "set"
REMOVE "variable" FROM "set"
LET "variable" = "expression", "any number of control phrases separated by commas"
STORE "expression" IN "variable", "any number of control phrases separated by commas"
FOR "local variable" = (expression)(expression)(expression)
[FOR EACH / FOR ALL / FOR EVERY] "permanent entity" "local variable"
[FOR EACH / FOR ALL / FOR EVERY] "local variable" [OF / IN / ON / AT] "set"
WITH "expression" "comparison" "expression"
OR "expression" "comparison" "expression"
AND "expression" "comparison" "expression"
DO
LOOP
IF "expression" "comparison" "expression"
IF "set" [IS / IS NOT] EMPTY, "any statement"
GO TO "statement number"
FIND "variable" [MAX / MIN] OF "expression"
WHERE "variable"
FIND FIRST, "one or more control phrases and WHERE phrase", IF NONE, "any statement"
[EXOGENOUS / EXOG] EVENT "event name"
[ENDOGENOUS / ENDOG] EVENT "event name"
RETURN
END
SUBROUTINE "subroutine name" ("arguments, if any")
CALL "subroutine name" ("arguments, if any")
REPORT "report name"
STOP

Figure 4-1. SIMSCRIPT 1.5 Commands [4]

```
PREAMBLE
NORMALLY MODE IS REAL
EVENT NOTICES INCLUDE ARRIVAL AND STOP
   EVERY COMPLETION HAS A TASK
TEMPORARY ENTITIES
   EVERY JOB HAS AN ATIME, A RUNTIME, A PABORT
          AND MAY BELONG TO THE QUEUE
THE SYSTEM OWNS THE QUEUE
DEFINE QUEUE AS A FIFO SET
DEFINE NUMTERM,TERMS,DONE,ABORT,BUSY AS INTEGER VARIABLES
DEFINE PABORT AS AN INTEGER VARIABLE
DEFINE TOT.TIME AS A DUMMY VARIABLE
TALLY SERVICE.TIME AS THE MEAN OF TOT.TIME
TALLY QUEUE.SIZE AS THE MEAN OF N.QUEUE
END


MAIN
''
'' .............................
'' SIMSCRIPT SINGLE CPU EXAMPLE .
'' .............................
''
LET TERMS=100                        ''  RUN 100 JOBS TO COMPLETION...
LET NUMTERM = 0
LET DONE = 0                         ''  INITIALIZR COUNTERS...
LET ABORT = 0
LET BUSY = 0
''
SCHEDULE AN ARRIVAL NOW              ''  INITIAL JOB ARRIVAL...
''
START SIMULATION
''
PRINT 1 LINE WITH DONE,ABORT THUS
#DONE: ***     #ABORTED:  **
PRINT 2 LINES WITH SERVICE.TIME,QUEUE.SIZE THUS
MEAN JOB THROUGHPUT TIME = ***.** SECONDS
MEAN QUEUE SIZE = ***.**
STOP
END
```

- SIMULATION OUTPUT -

```
#DONE:   92      #ABORTED:    8
MEAN JOB THROUGHPUT TIME =1663.07 SECONDS
MEAN QUEUE SIZE =   31.60
```

Figure 4-2a. SIMSCRIPT Example

attributes: arrival time (ATIME), execution/abort time (RUNTIME), and an abort indicator (PABORT). While GPSS handles queues invisibly and automatically, SIMSCRIPT users must define them explicitly as fifo-ordered sets, and use the FILE and REMOVE statements to insert and delete elements from the set. The TALLY statements enable SIMSCRIPT data gathering techniques for variable TOT-TIME which represents the time each job is resident in the computer system, and N.QUEUE, the size of the jobs queue.

The MAIN program sets up and initiates the actual simulation experiment. The number of job terminations (TERMS) is set to 100 to match the earlier GPSS run of a similar model. While GPSS will (START 100) automatically stop after 100 terminates, we must count them ourselves and stop the simulation ourselves. GPSS block counts show the number of transactions through each block (statement). To reflect jobs done vs. those aborted, we will run our own counters DONE and ABORT.

We then direct the initial job arrival to commence and we are prepared to START SIMULATION. Control passes to the most imminent event routine (ARRIVAL in our case since it is the only active event to occur so far) and the simulation proceeds, always handling the next imminent event and updating TIME.V accordingly. When no more future events exist, the simulation stops, and control passes back to the main program. We output our results showing the actual number of job completions and job aborts. Service time is computed and output as directed (the average of TOT.TIME) and the simulation halts.

Now that we have specified the PREAMBLE and MAIN, only the events routines remain. Figure 4-2b illustrates the SIMSCRIPT code for job arrival, job completion, and simulation stop. The strength of SIMSCRIPT is that each event can be defined individually--as the model complexity increases, the modelers coding stays relatively constant. None that event STOP simply prevents the next job from occurring, so the simulation will stop when the current jobs are completed.

```
        EVENT ARRIVAL SAVING THE EVENT NOTICE
        ''
        CREATE JOB
        LET ATIME = TIME.V
        ''
        LET KILL = UNIFORM.F(0.,1.,1)
        IF KILL>.1  LET RUNTIME=UNIFORM.F(66.,110.,1)
                    LET PABORT=0
                    GO TO RUN
           ELSE LET RUNTIME=UNIFORM.F(1.,5.,1)
                LET PABORT = 1
        ''
    'RUN' IF BUSY=1  FILE JOB IN QUEUE
                     GO TO NEXT
           ELSE LET BUSY=1
                SCHEDULE A COMPLETION(JOB) AT TIME.V+RUNTIME
        ''
   'NEXT' RESCHEDULE AN ARRIVAL AT TIME.V+UNIFORM.F(40.,60.,1)
        RETURN
        END



        EVENT COMPLETION(JOB)
        ''
        LET TOT.TIME=TIME.V-ATIME
        IF PABORT=1  LET ABORT = ABORT+1
           GO TO QUIT
           ELSE LET DONE = DONE+1
        ''
    'QUIT' DESTROY JOB
        LET NUMTERM = NUMTERM+1
        IF NUMTERM > TERMS-1  SCHEDULE A STOP NOW
                              RETURN
        ''
        ELSE IF QUEUE IS EMPTY  LET BUSY=0
                                RETURN
        ''
        ELSE REMOVE THE FIRST JOB FROM QUEUE
             SCHEDULE A COMPLETION(JOB) AT TIME.V+RUNTIME
        RETURN
        ''
        END



        EVENT STOP
        ''
        CANCEL THE ARRIVAL
        DESTROY THE ARRIVAL
        RETURN
        ''
        END
```

Figure 4-2b. SIMSCRIPT Example

Event ARRIVAL is executed for every new job. This is done in the typical bootstrapping method by scheduling the arrival of the next job (statement 'NEXT') for the future. If we know the inter-arrival time of an event, we can sample from this distribution and always keep one step ahead of ourselves. This is preferable to creating a list of all arrivals (100 elements in our case) and then handling them at their appointed times. The KILL variable helps make the job abort decision for 10 percent of the jobs, just as we did in the GPSS model.

If the cpu is busy, then this job must be enqueued until the cpu becomes free. This is automatic in GPSS; it is more explicit in SIMSCRIPT. We set up the BUSY variable to represent the permanent entity cpu, with a 1 value to denote "in-use". The GPSS invisible queue is also explicit in SIMSCRIPT. In the PREAMBLE we set up QUEUE as a first-in first-out SET to simulate a queue for the waiting jobs. Thus, an incoming job either grabs the cpu and thus generates its projected completion event or steps into the waiting line. In either case we schedule the next job arrival before returning control to the SIMSCRIPT event control module.

The next event to occur now is either completion of the first job or arrival of the second job. Whenever the most imminent event is job completion, event COMPLETION is called, and the JOB that just completed, with attributes ATIME, RUNTIME, and PABORT is sent along as the SIMSCRIPT-defined parameter. We update the appropriate counter to show jobs done or aborted. We destroy this temporary entity and its associated attribute storage to avoid eventual storage overflow as this storage can now be reused. If our stop condition has been raised, then we invoke event STOP right now to stop our bootstrapping of exogenous jobs.

Lastly, we do our own bootstrapping if there are other jobs waiting for the cpu, taking the job at the front of the queue and scheduling it to complete RUNTIME units of time from the present. We correctly computed this attribute in event ARRIVAL and carried it along with this job so that it could be used now rather than re-computed.

That completes our description of the SIMSCRIPT high-level model of a single-server computer system. Note the cause and effect relations of the SIMSCRIPT user-defined events. The focus is entirely upon event definition, with the supporting dynamic data structures and simulation controls in the background.

The output of this model is consistent with (although not identical to) the output of the earlier GPSS model. This is due to system randomness and the fact that the random number generators differ. However, if we ran both models many times and performed analyses, it could be shown that they indeed are correct characterizations of our single cpu system.

Our final example language is GASP, which can be used for continuous/discrete simulation. While the focus is, like with SIMSCRIPT, upon events, the underlying data structures must be apparent to the user or he/she cannot effectively model in GASP.

5. GASP

The GASP IV simulation language [6] was developed at Purdue University in 1970 and is currently maintained by Pritsker and Associates of West Lafayette, Indiana. It has historical predecessors in GASP II (Arizone State University) and the original GASP developed at U.S. Steel by Mr. Philip Kiviat. GASP is not so much a language as are GPSS and SIMSCRIPT; rather, it is a Fortran simulation interface--a set of some 24 routines to support additional user-written subroutines for simulation.

Since GASP is Fortran based, it is machine independent while SIMSCRIPT and GPSS are machine independent. Furthermore, GASP is modular, and easily modified or extended. It is the only well-documented language that supports both continuous and discrete simulation. Our comments herein will refer to only the discrete portions of GASP, as that is our central domain of interest.

5.1 Language Constructs

The GASP world view is similar to that of SIMSCRIPT: entities (people, equipment, orders, materials) and attributes (arrival rate, peak processing speed, quantity, type) are the basic building blocks of the simulation. Entities are grouped together into files which are similar to SIMSCRIPT sets. The dynamics of the simulation are effected by initializing the state of the system and going through all state transitions (change in entity-attribute values) to completion: the final state.

GASP is an activity-oriented simulation language, where each activity is defined as an ordered sequence of events. Events are described in terms of the mechanism by which they are scheduled. Time events are triggered by the clock; state-events are triggered when the system arrives at a particular state. The former are normally associated with discrete-event simulation, the latter with continuous simulation.

GASP provides an environment and Fortran interface to the modeler which has these functional capabilities:

(1) Event control
(2) State variable updating
(3) System state initialization
(4) Program monitoring and event reporting
(5) Interface to user-defined Fortran routines
(6) System performance data collection
(7) Statistical computations
(8) Random deviate generation, and
(9) Report generation.

Figure 5-1 illustrates the GASP system, composed GASP IV support routines and user-defined routines to handle the following:

(1) Initialization (MAIN and INTLC)
(2) State variables and state events (STATE and SCOND)
(3) Event code definitions (EVNTS) which calls the
(4) Event processing routines (user-named)
(5) Data collection and reporting (SSAVE and OTPUT), and
(6) Error handling (UERR).

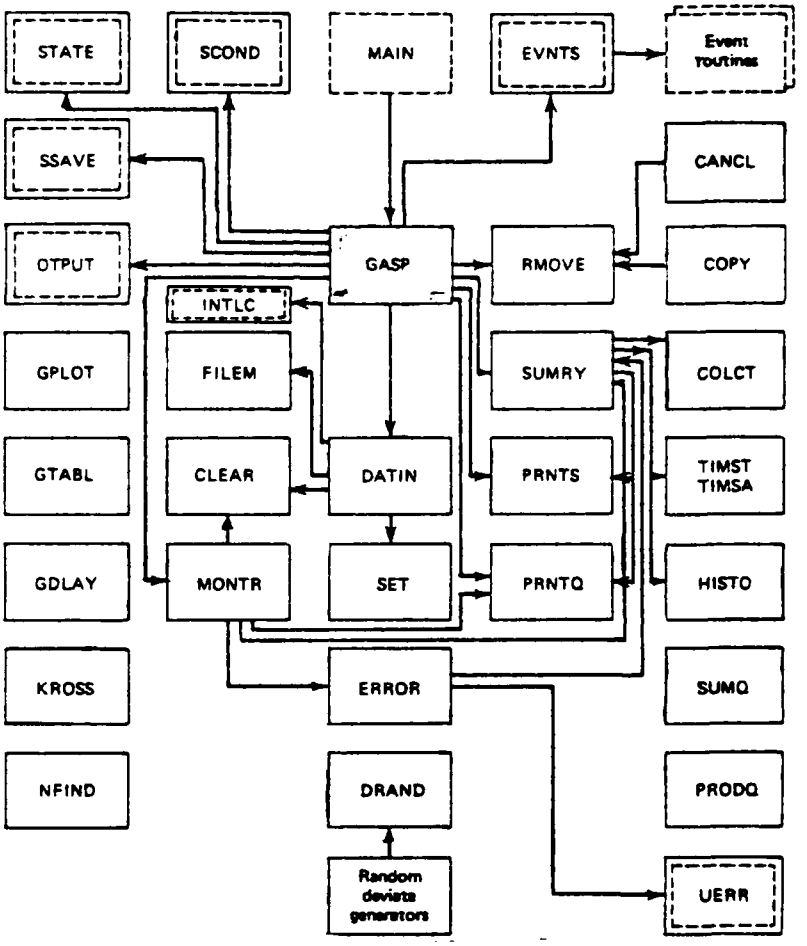


Figure 5-1. Relation of GASP IV and user subprograms. GASP IV subprogram. Dummy subprogram. User programs required. A → B Subprogram A calls subprogram B[5].

GASP can be utilized to model the same types of situations as do GPSS and SIMSCRIPT. But recall that GPSS is network or process-oriented whereas SIMSCRIPT and GASP are event-oriented. The modeler cannot possibly create a GASP-based simulation without a sound knowledge of the Fortran language.

5.2 Single Server Model

The Figures 5-2 (a through c) demonstrate a GASP simulation of our example system of Figure 2. Although this model appears to be much longer and more complex than those of the predecessor examples, it is structurally equivalent to the SIMSCRIPT model just discussed. Hence, we will discuss the GASP model in light of the SIMSCRIPT model of Figures 4-2 (a and b).

The SIMSCRIPT language provides a flexible Preamble section for modeler data structures to describe entities and attributes. In GASP these structures are realized as Fortran arrays and pre-defined in GASP labeled common blocks. As can be seen in the Fortran mainline comments, four attributes will be utilized to describe each job passing through the system, denoting respectively: event time, event type, arrival time, and probability of job abort. User variables representing the number of job runs/aborts, the cpu facility, and stopping criterion (100 jobs) complete the mainline GASP definition. Control is passed to GASP to control the entire simulation.

The first thing the GASP simulator does is initialization: user inputs define variable limits and structure sizes; routine INTLC is called to enable user initialization at the start of each run of the simulation. In our example we wish to start the simulation with a job arrival in the first 50 $\pm$ 10 seconds. So an entity is created with attributes detailing an arrival at time 50 $\pm$ 10 second from simulation start with some probability to abort. We place this in the GASP events queue and also trigger the statistics collector or cpu utilization (TIMST), as we know the cpu is initially free.

```
      PROGRAM MAIN(INPUT,OUTPUT,TAPE5,TAPE6=OUTPUT)
C     ..........................
C     .  GASP MAIN PROGRAM     .
C     ..........................

      DIMENSION NSET(3500)
      COMMON QSET(3500)
      COMMON/GCOM1/ ATRIB(25),JEVNT,MFA,MFE(100),MLE(100),MSTOP,
     +NCRDR,NNAPO,NNAPT,NNATR,NNFIL,NNQ(100),NNTRY,NPRNT,
     +PPARM(50,4),TNOW,TTBEG,TTCLR,TTFIN,TTRIB(25),TTSET
      COMMON/USER/ NJOBS,CPU,ABORTS,RUNS
      INTEGER FREE,CPU,ABORTS,RUNS
      EQUIVALENCE (NSET(1),QSET(1))
      DATA FREE/0/
C     ..................................................
C     .  SIMULATION OF A SINGLE-SERVER COMPUTER SYSTEM  .
C     .                                                 .
C     .  ..ATTRIBUTE(1) = TIME OF EVENT                 .
C     .  ..ATTRIBUTE(2) = EVENT CODE                    .
C     .  ..ATTRIBUTE(3) = JOB ARRIVAL TIME              .
C     .  ..ATTRIBUTE(4) = PR(ABORT) +++ ABORT/RUN       .
C     ..................................................

C     ...INITIALIZE GASP PARAMETERS...
      NCRDR=5
      NPRNT=6

C     ...INITIALIZE USER RUN PARAMETERS...
      ABORTS=0
      RUNS=0
      CPU=FREE
      NJOBS=100

C     ...GIVE CONTROL TO THE GASP SIMULATOR NOW...
      CALL GASP
      STOP
      END
```

```
      SUBROUTINE INTLC
C     ................

      COMMON/GCOM1/ ATRIB(25),JEVNT,MFA,MFE(100),MLE(100),MSTOP,
     +NCRDR,NNAPO,NNAPT,NNATR,NNFIL,NNQ(100),NNTRY,NPRNT,
     +PPARM(50,4),TNOW,TTBEG,TTCLR,TTFIN,TTRIB(25),TTSET
      INTEGER EVENTS,ARRIVE,CPUTIL
      DATA EVENTS/1/,ARRIVE/1/,CPUTIL/1/

C     ...CREATE FIRST ARRIVAL...
      ATRIB(1) = TNOW+DRAND(1)*20+40
      ATRIB(2)=ARRIVE
      ATRIB(4)=DRAND(1)
      CALL FILEM(EVENTS)

C     ...INITIALIZE STATISTICS GATHERING...
      CALL TIMST(0.,TNOW,CPUTIL)
      RETURN
      END
```

Figure 5-2a. GASP Example

```
      SUBROUTINE EVNTS(CODE)
C     ......................
C
      INTEGER CODE
C     ...CARRY OUT USER-DEFINED EVENT...
      GO TO (1,2,3),CODE
C     ....
    1 CALL ARRIVAL
      RETURN
C     ....
    2 CALL COMPLTN
      RETURN
C     ....
    3 CALL STOP
      RETURN
      END


      SUBROUTINE ARRIVAL
C     ..................
C
      COMMON/GCOM1/ ATRIB(25),JEVNT,MFA,MFE(100),MLE(100),MSTOP,
     +NCRDR,NNAPO,NNAPT,NNATR,NNFIL,NNQ(100),NNTRY,NPRNT,
     +PPARM(50,4),TNOW,TTBEG,TTCLR,TTFIN,TTRIB(25),TTSET
      COMMON/USER/ NJOBS,CPU,ABORTS,RUNS
      INTEGER CPU,ABORTS,RUNS,CMPLETE,ARRIVE,CPUTIL
      INTEGER BUSY,ABORT,RUN,RUNTIME,EVENTS,QUEUE
      DATA BUSY/1/,ABORT/1/,RUN/0/,EVENTS/1/,QUEUE/2/,
     +CMPLETE/2/,ARRIVE/1/,CPUTIL/1/
C
C     ...RECORD JOB ARRIVAL AND SEIZE CPU IF FREE...
      ATRIB(3) = TNOW
      IF(CPU .EQ. BUSY) GO TO 3
      CPU = BUSY
      CALL TIMST(1.,TNOW,CPUTIL)
      IF(ATRIB(4) .GT. .1) GO TO 1
C
C     ...CALCULATE RUN/ABORT TIME...
      ATRIB(4) = ABORT
      RUNTIME=DRAND(1)*4+1
      GO TO 2
    1 ATRIB(4) = RUN
      RUNTIME = DRAND(1)*44+71
C
C     ...SCHEDULE THE SUBSEQUENT JOB COMPLETION...
    2 ATRIB(1) = TNOW+RUNTIME
      ATRIB(2) = CMPLETE
      CALL FILEM(EVENTS)
      GO TO 4
C
C     ...FILE JOB IN QUEUE TO AWAIT CPU FREE...
    3 CALL FILEM(QUEUE)
C
C     ...AND ALWAYS SCHEDULE NEXT ARRIVAL...
    4 INTRAVL = DRAND(1)*20+40
      ATRIB(1) = TNOW+INTRAVL
      ATRIB(2) = ARRIVE
      ATRIB(4) = DRAND(1)
      CALL FILEM(EVENTS)
      RETURN
      END
```

Figure 5-2b. GASP Example

```
      SUBROUTINE COMPLTN
C     ..................
      COMMON/SCOM1/ ATRIB(25),JEVNT,MFA,MFE(100),MLE(100),MSTOP,
     +NCRDR,NNAPO,NNAPT,NNATR,NNFIL,NNQ(100),NNTRY,NPRNT,
     +PPARM(50,4),TNOW,TTBEG,TTCLR,TTFIN,TTRIB(25),TTSET
      COMMON/USER/ NJOBS,CPU,ABORTS,RUNS
      INTEGER FREE,EVENTS,CMPLETE,QUEUE,EMPTY,CPU,ABORTS,RUNS
      INTEGER ABORT,RUN,THRUPUT,CPUTIL,STOP
      DATA FREE/0/,EVENTS/1/,CMPLETE/2/,QUEUE/2/,EMPTY/0/
      DATA ABORT/1/,RUN/0/,THRUPUT/1/,CPUTIL/1/,STOP/3/
C
C     ...STATS ON THROUGHPUT TIME,RUNS,ABORTS...
      TOTIME = TNOW-ATRIB(3)
      CALL COLCT(TOTIME,THRUPUT)
      IF(ATRIB(4) .EQ. ABORT) ABORTS = ABORTS+1
      IF(ATRIB(4) .EQ. RUN) RUNS = RUNS+1
      NJOBS = NJOBS-1
C
C     ...SCHEDULE SIMULATION STOP IF APPROPPIATE...
      IF(NJOBS .GT. 0) GO TO 1
      ATRIB(1) = TNOW
      ATRIB(2) = STOP
      CALL FILEM(EVENTS)
      RETURN
C
C     ...BOOTSTRAP IF JOB AWAITING CPU...
    1 IF(NNQ(QUEUE) .EQ. EMPTY) GO TO 4
      CALL RMOVE(MFE(QUEUE),QUEUE)
      IF(ATRIB(4) .GT. .1) GO TO 2
C
C     ...CALCULATE RUN/ABORT TIME...
      ATRIB(4) = ABORT
      RUNTIME = DRAND(1)*4+1
      GO TO 3
    2 ATRIB(4) = RUN
      RUNTIME = DRAND(1)*44+71
C
C     ...SCHEDULE JOB COMPLETION..
    3 ATRIB(1) = TNOW+RUNTIME
      ATRIB(2) = CMPLETE
      CALL FILEM(EVENTS)
      RETURN
C
C     ...ELSE RELEASE CPU & UPDATE STATS...
    4 CPU = FREE
      CALL TIMST(0.,TNOW,CPUTIL)
      RETURN
      END



      SUBROUTINE STOP
C     ...............
      COMMON/SCOM1/ ATRIB(25),JEVNT,MFA,MFE(100),MLE(100),MSTOP,
     +NCRDR,NNAPO,NNAPT,NNATR,NNFIL,NNQ(100),NNTRY,NPRNT,
     +PPARM(50,4),TNOW,TTBEG,TTCLR,TTFIN,TTRIB(25),TTSET
      COMMON/USER/ NJOBS,CPU,ABORTS,RUNS
      INTEGER CPU,ABORTS,RUNS
C
C     ...OUTPUT USER STATS...
      WRITE(NPRNT,100) ABORTS,RUNS
  100 FORMAT(15HNO. OF ABORTS =,I5,5X,13HNO. OF RUNS =,I5)
C
C     ...KILL THE SIMULATION...
      MSTOP = -1
      RETURN
      END
```

Figure 5-2c. GASP Example

GASP maintains the events queue, and automatically removes the front entry, which represents the most imminent event. The clock (TNOW) is updated to the first attribute of the entity and the event code (CODE) is taken from the entities second attribute. Control is passed to user-defined routine EVNTS which simply invokes the proper service routine dependent upon the event code.

This is the essence of a GASP discrete-event simulation: the modeler identifies and codes routines to represent the salient events of the system. He also places into the GASP events queue entities representing future events which define the sequence of activities to be performed. One by one, GASP updates the clock, passes control to the event routine, and so on until simulation stop time. As with our SIMSCRIPT model, these events describe the model's activities: job arrivals, job completions, and simulation stop. An independent user routine is written for each event description.

Let us consider the job arrival event as defined by routine ARRIVAL. (Note that it is equivalent to the same event routine in the SIMSCRIPT example.) We record the arrival time as an attribute for later statistics purposes and seize the cpu if possible. If the cpu is available, we determine and schedule a later job completion; if not, then we place the job in a wait queue. In either case we bootstrap the arrivals by scheduling the next arrival to occur.

Job completions trigger a call to routine COMPLTN. We calculate this job's elapsed time in the system and ask GASP to cumulate this data (routine COLCT). We then update our own counts of job runs/aborts and determine if the 100th job has completed yet (GPSS does these things automatically). If so, we schedule a stop now and return control to the GASP monitor. Otherwise, we see if another job awaits the cpu, and if so we remove it from the wait quque and schedule a job completion at the appropriate time. If no job awaits the cpu, then we set it free and tell the statistics support routine (TIMST) of this change of status.

Lastly, routine STOP handles the simulation halt event by reporting the number of runs/aborts (in a fashion similar to that of the SIMSCRIPT statements following the START SIMULATE statement of the main program) and then setting a particular switch called MSTOP. The GASP system, in response to this, immediately shuts down event handling and finalizes the statistics of the system (queue, etc.) and the user (cpu use and throughput time), outputs them and halts. Any events scheduled for the future are shown as awaiting but not carried out.

The GASP output (Figure 5-3) agrees well with our earlier GPSS and SIMSCRIPT models of the single server system. Approximately 10 percent of the 100 jobs aborted as expected. Mean job throughput time (average time job was in the system) of 1,641 agrees with SIMSCRIPT's value of 1,663 more than GPSS's value of 1,473. Lastly, our GASP mean queue size of 32.2 is close to SIMSCRIPT's 31.6 and GPSS's 27.3. As we noted earlier, many runs of longer duration would be required before we could place any statistical confidence in these results. However, it is clear that all three models show a clogged system, with 50 or more jobs awaiting the cpu after 100 jobs have executed, and the data concerning utilization, etc. are all of similar magnitudes giving credence to model similarity.

GASP has undergone several major updates and is a stable, portable simulation aid. Its shortcoming (and that of SIMSCRIPT also) lies in its inability to easily describe process-oriented systems (this is, of course, GPSS's forte). The major GASP advantage is that combined event oriented and continuous simulations can be carried out, which no other language enables. In this light, GASP also enables activity-oriented simulation (in addition to event oriented) through the use of its "state variables".

GASP SUMMARY REPORT

SIMULATION PROJECT NUMBER 117 BY L.L.ROSE

DATE 12/ 4/ 1979 RUN NUMBER 1 OF 1

CURRENT TIME = .8317E+04

STATISTICS FOR VARIABLES BASED ON OBSERVATION

	MEAN	STD DEV	SD OF MEAN	CV	MINIMUM	MAXIMUM
THRUPUT	.1641E+04	.9549E+03	.9549E+02	.5819E+00	.4000E+01	.3370E+04

STATISTICS FOR TIME-PERSISTENT VARIABLES

	MEAN	STD DEV	MINIMUM	MAXIMUM	TIME INTERVAL	CUR. VALUE
CPU USE	.9860E+00	.1174E+00	0.	.1000E+01	.8317E+04	.1000E+01

INTERMEDIATE RESULTS

NO. OF APORTS = 12 NO. OF RUNS = 88

PRINTOUT OF FILE NUMBER 2
TNOW = .8317E+04
OOTIM= .8267E+04

TIME PERIOD FOR STATISTICS	.8317E+04
AVERAGE NUMBER IN FILE	32.1822
STANDARD DEVIATION	19.2494
MAXIMUM NUMBER IN FILE	66

Figure 5-3. GASP Example Results

5.3 SLAM

Pritsker has recently announced a very strong extension to GASP, a new language called SLAM: A Simulation Language for Alternative Modeling [7]. In essence, he has added a network or process-oriented front end to the GASP simulator. The end result is a capability to conveniently model process-oriented or event-oriented or continuous systems all under the umbrella of a single language, SLAM. The power of SLAM lies in the fact the portions of a model may be described using process-oriented statements; other portions of the same model may be defined in an event or continuous-oriented manner. This enables the modeler to describe the system in a more natural manner and does not force a system views upon the model definition.

Time does not permit a full discussion of the continuous simulation features of GASP/SLAM or of the process/network GPSS-like features of SLAM. Certainly, though, we expect SLAM, with the GASP base, to be a strong contributor to simulation in the future.

6. GENERAL SIMULATION LANGUAGES: APPLICABILITY

The objective of the previous five chapters was to define the basics of simulation and to introduce the three major general simulation languages: GPSS, SIMSCRIPT, and GASP. An example was carried out in all three languages to demonstrate applicability to computer systems simulation and to show the main features of each language.

We have observed some of the powers of a simulation language: recursiveness (a completion/arrival scheduling another completion/arrival) even though the supporting language is usually strictly non-recursive. The fact that many events can co-occur at any moment provides us the property of parallel processes, even though digital computers are strictly serial in execution. All three subject languages exhibit these powerful properties. Further, they all provide a host

environment suitable for simulation. In particular, we note the presence of queueing capabilities (a most important simulation construct) and timing capabilities for handling the traversal of time in coordination with event occurrences. Lastly, each language enables easy report generation and statistics collection to help the modeler analyze the system so defined.

Given this background, the reader is in a much stronger position to appreciate and understand the difficulty of simulation in general and of simulating computer systems in particular. What is required is a concise evaluation of these three languages in light of USACSC needs in the simulation area.

6.1 Selection of Criteria

Criteria for determining the applicability of a simulation language have been cited by this author [8], Pritsker [7], and USACSC personnel. Previous examination by this author in the computer systems/database area arrived at the following general and DBMS-oriented sets of criteria:

General	DBMS Oriented
1. Ease of use	1. Hardware characterization
2. User knowledge	2. Software characterization
3. Model flexibility	3. Data definition facility
4. Output statistics	4. Data manipulation facility
5. Program product	5. Device media control facility
6. Portability	

If we consider these criteria in light of simply computer systems simulation (with less emphasis than the previous study on DBMS), then one would prefer to replace the DBMS-oriented criteria with the following computer systems-oriented criteria to augment the general criteria:

(a) Hardware characterization

(b) Software characterization

(c) Database characterization

(d) Network characterization

(e) Operating system characterization.

Pritsker and Pegdon, in their new text on SLAM [7] offer the general criteria for simulation language determination as shown in Figure 6-1.

TRAINING REQUIRED	EASE OF LEARNING THE LANGUAGE
	EASE OF CONCEPTUALIZING SIMULATION PROBLEMS
CODING CONSIDERATION	EASE OF CODING INCLUDING RANDOM SAMPLING AND NUMERICAL INTEGRATION
	DEGREE TO WHICH CODE IS SELF-DOCUMENTING
PORTABILITY	LANGUAGE AVAILABILITY ON OTHER OR NEW COMPUTERS
FLEXIBILITY	DEGREE TO WHICH LANGUAGE SUPPORTS DIFFERENT MODELING CONCEPTS
PROCESSING CONSIDERATIONS	BUILT-IN STATISTICS GATHERING CAPABILITIES
	LIST PROCESSING CAPABILITIES
	ABILITY TO ALLOCATE CORE
	EASE OF PRODUCING STANDARD REPORTS
	EASE OF PRODUCING USER-TAILORED REPORTS
DEBUGGING & RELIABILITY	EASE OF DEBUGGING
	RELIABILITY OF COMPLIERS, SUPPORT SYSTEMS, & DOCUMENTATION
RUN-TIME CONSIDERATIONS	COMPLIATION SPEED
	EXECUTION SPEED

Figure 6-1. Features on Which to Evaluate a Simulation Language [7]

Lastly, USACSC personnel have provided guidance in a working paper entitled "Modeling and Simulation Tool Evaluation Criteria", which is reproduced verbatim herein:

Modeling and Simulation Tool
Evaluation Criteria

1. Ease of Use - Amount of human time, computer time or storage required to set up a model run? Can models be executed by specification of library components? Does the language offer the ability to easily alter hardware and workload parameters using interactive techniques? Does the product facilitate automated workload characterization via a job accounting system interface?

2. Modeler Knowledge and Understanding - How much modeler knowledge of discrete event and probabilistic simulation techniques; computer systems architecture; and database processing concepts is required to understand and use the modeling tool? Also, how much knowledge of the functional aspects of application systems is required? Must a detailed study of "what" a system is to accomplish precede the modeling and simulation activity in order to insure model accurace? Also, how long does it take to learn and become proficient in the use of the language? A range of timeframes based on personnel education, ADP experience, and modeling and simulation experience should be developed? Also, how many pages of documentation must be read and familiarized?

3. Model Flexibility - This factor addresses the long-term overall value of the simulation tool. How many different situations, e.g., DBMS, configuration variability, interactive processing, network architecture, etc., can the model represent without requiring major modification or enhancement? Also is the model's framework sufficiently general and modularized so as to facilitate the implementation of as yet unknown future systems' requirements?

4. Output Facilities - Does the output data completely and accurately describe the processes simulated in terms of utilization and wait queue statistics at the device level? Is it possible to capture and display data in various user-defined aggregates during the simulated processes? Is the output report data readable and clearly identified? Is it possible to interface with the simulation process so as to perform additional data collection and statistical manipulations as required?

5. Product Development Stage - Is the modeling and simulation tool a proven product? Has it been adequately developed and does it have a satisfied user community? Is the product kept up to date so as to be able to model and simulate contemporary information system concepts? Available documentation must be of sufficient clarity and detail to describe the product. User oriented published documentation should include a detailed guide showing in detail how to model and simulate using the tool.

included also should be a series of examples of actual models built and simulations executed using the product. These examples should clearly demonstrate the full capabilities of the product. The guide should not only describe user input specifications, but also should explain the meaning of all output statistics and yield some insight into how the data is collected, manipulated and output. Detailed documentation covering the detailed facets of the model's internal functions must be sufficient to permit user developed enhancements and special purpose interfaces with the model.

6. Portability - The portability of a software product relates to how easily it can be moved from one hardware system to another. Any machine dependent languages or other limiting factors is considered undesireable. A transportable product language such as ANSI FORTRAN or COBOL would be desireable.

From this background we can feel assured that our selection of general criteria will provide a proper basis for simulation language evaluation. Further criteria in the computer systems area as suggested earlier will be utilized as that is our area of simulation interest. The following notes will amplify these additional five criteria.

Computer Systems-Oriented Criteria

1. Hardware Characterization - Capability to characterize, at a general or specific level, standard computer components and their interconnections: cpu, device controller, channel, memory, disc, tape, drum, ecs, tty with regard to data transmission, flow, and control.
2. Software Characterization - Capability to characterize, at a general or specific level, user software programs and system software support/utility packages regarding hardware resource requirements/utilization.
3. Database Characterization - Capability to characterize the relevant components of the database technology: the data schema, system implementation, and data manipulation primitives available to the software component.
4. Network Characterization - Capability to model an arbitrary system of connected hardware elements: N1 cpu's, N2 discs, N3 channels, etc., so that flexible hardware designs can be modeled.

5. Operating System Characterization - Capability to model the computer system command and control center and its handling of resource requests and data flow among users and devices.

Given these criteria, both general and computer systems oriented, let us evaluate the three general simulation languages with regard to their utility to USACSC in the areas of general modeling and computer systems modeling.

6.2 General Evaluation

Ease of Use

GPSS is extremely easy to use and learn, in contrast to SIMSCRIPT or GASP. Models can be built in a matter of minutes, rather than days or weeks. The language is smaller, simpler, and requires less sophistication of the modeler. Added outputs, etc., within the pre-defined bounds of GPSS are almost trivial to incorporate into existing models. All three languages execute in reasonable, but long run times. GPSS has essentially no capability for modularization. Its concept of a subroutine is a primitive method for transfer of control to a non-sequential set of statements--no localization or formal parameterization is possible. Both SIMSCRIPT and GPSS are modular in that events are characterized by independent subroutines; these could be placed in a library for use by reference. GASP does offer an interactive version, but all three languages are essentially batch oriented. Both GASP and SIMSCRIPT require sophisticated programmer background and knowledge before they can be used effectively.

Rating:	GPSS	10	
	SIMSCRIPT	6	(scale of 10)
	GASP	5.	

Modeler Knowledge and Understanding

From a model creation standpoint, GPSS requires less knowledge as it is a simple modeling language, wheras SIMSCRIPT and GASP are computational modeling languages, with much more power, but also with debugging problems associated with the more sophisticated languages. Given a constructed model, SIMSCRIPT is the most self-documenting... GPSS the least. Knowledge of the system to be modeled is a separate issue from language considerations, although it can affect the desired level of detail of the model. A very high-level model would clearly favor GPSS, unless it is desired to slowly add detail to the model. The critical elements of understanding are those to be modeled in high detail; normally one does not desire to model the entire system in minute detail as that would require too much time, money, and personnel resources. SIMSCRIPT and GASP are more self-documenting than is GPSS.

Rating:	SIMSCRIPT	8	
	GASP	7	(scale of 10)
	GPSS	5.	

Model Flexibility

Much of the flexibility in any simulation model must be built into the initial design or it cannot be easily added later. All three languages carry attributes with each transaction, and these can easily be extended or altered. From a language standpoint, GPSS is much less flexible in that inputs and outputs are constrained to the language definition (the GPSS interpreter is not extensible). However, both GASP and SIMSCRIPT exhibit a high degree of flexibility and can offer a highly parameterized model of routines applicable to a wide area of modeling. GASP also enables continuous modeling to coexist with discrete: a strong feature for model flexibility.

Rating:	GASP	8	
	SIMSCRIPT	7	(scale of 10)
	GPSS	4.	

Output Facilities

Data collection and representation is a fundamental part of all simulation languages. Those of GPSS are most easily used by the modeler, but cannot be tailored. GASP and SIMSCRIPT facilities exist and can also be extended or tailored to suit the users needs; GASP is slightly more difficult as the user must work within the GASP data structures.

Rating:	SIMSCRIPT	10	
	GASP	9	(scale of 10)
	GPSS	7.	

Product Development Stage

All three of these languages are products of more than 10 years of development and testing and production. They have been utilized by hundreds of modelers and are stable, well-documented systems. Textbooks and manuals are available for each language to provide a modeling methodology and language documentation.

Rating: No preference: (all good) 10

Portability

The GPSS language is interpretive, but the interpreter is written in IBM Assembler. Several non-IBM versions have been implemented (CDC, for one) but GPSS remains mainly an IBM product. As such, its portability is limited. SIMSCRIPT is a true language with its own compiler. Versions have been written for most of the large mainframes (in particular, IBM and Honeywell) and, thus, it can be considered partially portable. Lastly, GASP is a collection of standard Fortran routines and, thus, is considered completely portable.

Rating:	GASP	10	
	SIMSCRIPT	7	(scale of 10)
	GPSS	5.	

One can compute a cumulative general rating for each of the three subject languages by adding up the individual ratings. The results are as follows: SIMSCRIPT (48), GASP (45), and GPSS (45). It is clear why all three of these simulation languages enjoy extensive use--they rate nearly equal in a general modeling context and satisfy our criteria at the 75 percent level.

Our conclusion, then, is that for general simulation in the context of USACSC criteria, any one of these languages is suitable. Let us now extend our evaluation to the computer systems orientation and discuss how these languages can be used to simulate such systems.

6.3 Computer Systems-Oriented Evaluation

Hardware Characterization

None of these three languages has constructs to explicitly *describe hardware components*. It would boil down to a functional definition in each language, to compute time requirements. GASP, with its continuous simulation capability, could probably best model a moving head disk to most accurately portray i/o timings. These hardware component models, combined to detail flow and control, would be non-trivial to construct, and difficult to alter significantly.

Rating:	GASP	4	
	SIMSCRIPT	3	(scale of 10)
	GPSS	1.	

Software Characterization

Again, software modeling is simply alien to these simulation languages. This really requires a process orientation, as opposed to an event or network orientation as exhibited by these languages. Process description (by the user) would be extremely difficult in GPSS if even possible. Major extensions are required to overcome these deficiencies in all three candidate languages.

Rating:	SIMSCRIPT	4	
	GASP	3	(scale of 10)
	GPSS	1.	

Database Characterization

The notions of data independence and data structure and schema are far removed from the capabilities of our three languages. The host system must be designed with database in mind. GPSS has no concept of data structure whatsoever, GASP a little, and SIMSCRIPT the set construct which could be quite helpful.

Rating:	SIMSCRIPT	3	
	GASP	1	(scale of 10)
	GPSS	0.	

Network Characterization

Hardware components house software processes, and to flexibly characterize these is unduly difficult in these general languages. The protocols of message/packet switching, multiplexing, etc., become important and yet are nontrivial extensions to these languages. GPSS, in particular, is not equipped to handle a parameterized component by component definition followed by their interconnections.

Rating:	SIMSCRIPT	2	
	GASP	2	(scale of 10)
	GPSS	1.	

Operating System Characterization

A good model of an operating system is a difficult job in itself: reflecting the proper roles of job management, data management, resource management, and task management. Here a parameterized model

is clearly preferred--something the user can refine and utilize without undue strain. Otherwise a large, complex model in any of these languages would be required, whose run times would be expected to be very long. The stronger primitives for preempt and assemble favor GPSS in this case.

Rating:	GPSS	4	
	SIMSCRIPT	3	(scale of 10)
	GASP	2.	

Our specific ratings for language value for computer systems simulation are: SIMSCRIPT (15), followed by GASP (11) and GPSS (7). Most important is the fact that, on a scale of 50, our simulation ratings achieved 30 percent acceptance at best. This is unsatisfactory for several reasons. First, a 30 percent rating (for a costly language) means that there are serious and substantial deficiencies in the language constructs available to the modeler. Secondly, defining and implementing the model language constructs is a large task requiring money, time, and experts in simulation, of which there are few. Third, the simulation expert must also be a computer systems expert or an additional expert is required.

7. CONCLUSIONS

It is now clear that general simulation language tools such as GPSS, GASP, and SIMSCRIPT, while valuable modeling aids in a wide variety of application areas, lack the language and modeling constructs that would appreciably ease the job of the computer systems model developer or user. Major enhancements are required to bring any of these languages to the point where they can be applied effectively in this area; none of these general simulation languages is acceptable to USACSC for computer systems simulation. A more specialized and sophisticated tool is required.

Research in this area has been active, and several specialized computer systems simulation languages have recently been developed.

The Computer System Simulator (CSS) was developed at IBM in 1969; the Extended Computer Systems Simulator (ECSS) was developed by extending SIMSCRIPT by FEDSIM; the Information Processing Systems Simulator (IPSS) is a FORTRAN-based product that has been developed under NSF and AIRMICS support; a superset of GASP for computer systems called SIMTRAN has been developed at GE.

Each of these languages is preferable to the general languages GASP, GPSS, and SIMSCRIPT for computer systems simulation. They have, among other facets, the inherent capability to model process control and resource management: bare necessities for computer systems. Constructs particular to hardware characterization, etc., enable the modeler to focus upon the experiment and to construct self-defining simulation models.

A brief consideration of these languages for computer systems simulation would probably rule out CSS and SIMTRAN due to their propriety nature. IPSS is a public domain product and ECSS is available to any agency on government contract [9,10]. In earlier research for USACSC, this author evaluated IPSS in a database context [8]. Further evaluation documentation pertaining also to ECSS can be found in the FEDSIM evaluation performed early in 1978 [11]. These two documents should serve as a sound basis for any further comparison of the languages IPSS and ECSS.

On the surface, it appears that either of these languages (IPSS or ECSS) has the vital ingredients necessary to serve as the basic tool for USACSC computer systems simulation. Further research is needed to provide an overview of these computer systems-oriented simulation languages, their capabilities, and their utility to USACSC needs and requirements. Matters such as portability, price, and availability may affect the attractiveness of some of the candidate languages; this must be considered by the researcher to ensure the utility of any evaluation report on computer systems-oriented simulation languages.

This research has provided a firm foundation for USACSC understanding and evaluation of computer languages for computer systems simulation. Sound criteria have been established for language applicability determination; they must be used should further evaluation be desired. The three major languages for discrete event simulation have been overviewed and demonstrated to construct a basis for USACSC simulation of computer systems. More sophisticated, computer-oriented languages are required to meet these USACSC requirements. Several have been suggested as viable solutions to this complex problem.

Further training will be required of USACSC personnel so that these computer systems simulation languages can be used effectively. A blend of modeling skills, computer systems understanding, statistics, and language knowledge is necessary to do the task properly. Personnel can accomplish this through education, tutoring, self-instruction, and experience through model development with others knowledgeable in the area.

Although this is a non-trivial undertaking, it is nonetheless feasible and viable. It promises to greatly enhance USACSC capabilities in evaluation, prediction, and selection of computer systems. This author strongly believes that unless one is involved in model construction, the model cannot be utilized to its full potential. Hence, it is believed that this approach of simulation language selection for model construction by USACSC personnel rather than parameterized model selection will accrue the best benefits for USACSC in the long run.

8. REFERENCES

1. Shannon, R.E., Systems Simulation, Prentice Hall, Englewood, N.J., 387 pages, 1975.

2. Mihram, G.A., Simulation: Statistical Foundations and Methodology, Academic Press, New York, N.Y., 526 pages, 1972.

3. IBM Corporation, General Purpose Simulation System V, Users Manual, (Form SH-20-0851), White Plains, N.Y.

4. Gordon, G., System Simulation, Second Edition, Prentice Hall, Englewood Cliffs, N.J., 324 pages, 1978.

5. Kiviat, P.J., Villanueva, R., and Markowitz, H.M., SIMSCRIPT II-5 Programming Language, C.A.C.I., edited by Russell, E.C., Los Angeles, CA, 383 pages, 1973.

6. Pritsker, A., The GASP IV Simulation Language, Wiley-Interscience, New York, N.Y., 451 pages, 1974.

7. Pritsker, A., and Pegden, D., Introduction to Simulation and SLAM, Wiley and Sons, New York, N.Y., 588 pages, 1979.

8. Rose, L.L., "Computer Systems/Database Simulation", Final Report to AIRMICS (D.O. 0909), 113 pages, 1978.

9 DeLutis, T.G., "A Methodology for the Performance Evaluation of Information Processing Systems", Final Report to the National Science Foundation, OSIS, GN36622, 1977.

10. Kosy, D.W., The ECSS II Language for Simulating Computer Systems, R-1895-GSA, December, 1975, RAND, Santa Monica, CA.

11. "Evaluation of DBMS Modeling Approaches", FEDSIM Report MV-027-033-ARMY, February, 1978, Washington, D.C.

END

DATE
FILMED
10-81

DTIC