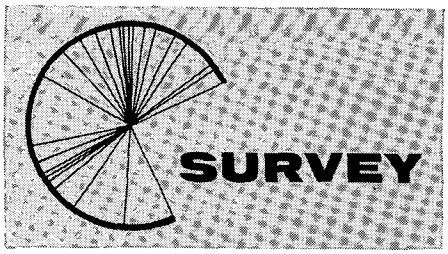


Computer Simulation—Discussion of the Technique and Comparison of Languages

By Daniel Teichroew* and John Francis Lubin†

The purpose of this paper is to present a comparison of some computer simulation languages and of some of the packages by which each is implemented. Some considerations involved in comparing software packages for digital computers are discussed in Part I. The issue is obvious: users of digital computers must choose from available languages or write their own. Substantial costs can occur, particularly in training, implementation and computer time if an inappropriate language is chosen. More and more computer simulation languages are being developed: comparisons and evaluations of existing languages are useful for designers and implementers as well as users.

The second part is devoted to computer simulation and simulation languages. The computational characteristics of simulation are discussed with special attention being paid to a distinction between continuous and discrete change models. Part III presents a detailed comparison of six simulation languages and packages: SIMSCRIPT, CLP, CSL, GASP, GPSS and SOL. The characteristics of each are summarized in a series of tables. The implications of this analysis for designers of languages, for users, and for implementers are developed.

The conclusion of the paper is that the packages now available for computer simulation offer features which none of the more general-purpose packages do and that analysis of strengths and weaknesses of each suggests ways in which both current and future simulation languages and packages can be improved.

This study was supported, in part, by funds made available by the Ford Foundation to the Graduate School of Business, Stanford University. However, the conclusions, opinions and other statements in this publication are those of the authors and are not necessarily those of the Ford Foundation.

A draft of this paper was prepared for the Workshop on Simulation Languages, Graduate School of Business, Stanford University, March 6 and 7, 1964. The paper has benefitted from suggestions from participants at the Workshop, particularly Michael Montalbano, and from projects carried out by students in the Graduate School of Business: H. Barnett, H. Guichelaar, Lloyd Krause, John P. Seagle, Charles Turk, Victor Preisser. The paper has also benefitted from discussions held in connection with the Workshop on Simulation Languages, University of Pennsylvania, March 17–18, 1966.

I. Comparison of Languages and Software Packages for Digital Computers

Programming a problem for digital computers has been aided considerably by the development of special purpose programs, assemblers, interpreters and higher level languages. The basic purpose of all these "languages," and the "software packages" that are provided to implement them (compilers, documentation, diagnostic aids, etc.), is to provide a definer of problems and his programmer with a means for communicating a problem to the machine without doing so in machine language. The packages take care of many of the mechanical details of programming: memory location assignments, input-output commands, and relationships among program segments. The languages also help the problem definer and the programmer communicate with each other.

The term "language" is used here to mean the specifications of the language without regard to implementation; the term "package" or "software package" means the set of materials available to the user: programs, manuals, debugging aids, etc. In a discussion of languages it is useful to distinguish three different activities: (*i*) the specification or design of a language, (*ii*) the implementation of the language in the form of a computer program and documentation, and (*iii*) the use of the language and the package. Sometimes the same individuals are involved in all three activities. It is convenient to speak of three groups: language designers, language implementers, and users.

The usefulness of languages and software packages has been great, and the number of them has increased rapidly. Frequently, more than one can be used for a particular problem. The user must then choose from among several competing packages, each of which may offer some features

The characteristics of the languages and software packages change rapidly. The statements in the paper were originally intended for the situation current in March, 1964. Where significant changes have occurred the text has been modified.

* Division of Organizational Sciences, Case Institute of Technology, University Circle, Cleveland, Ohio.

† Director for Computing Activities, University of Pennsylvania, Philadelphia, Pa.

CONTENTS

I. Comparison of Languages and Software Packages for Digital Computers
II. The Simulation Technique and Simulation Languages
  1. Simulation as a Technique for Problem Solving and Research
  2. Continuous-change Models
  3. Discrete-change Models
  4. Evaluation of Languages for Simulation
III. Comparison of Six Discrete Flow Simulation Packages
  1. Description of Six Languages
  2. Previous Comparisons
  3. Comparison of Six Simulation Packages
IV. Implications for Users, Language Implementers, and Language Designers
  1. Users
  2. Language Implementers
  3. Language Designers
V. References and Bibliography

suitable for his particular problem or class of problems. A comparison of existing languages and packages for a particular set of problems can be very helpful: language designers and implementers from such a comparison can determine features that should be incorporated into future versions; users can utilize the comparison to help in selecting a particular language for particular applications.

As a practical matter, the use of a language is determined not only by the properties of the language itself but also by the characteristics of the package and by features of the programming systems available for particular equipment. These characteristics include the range of machines for which the language is implemented; the relationship of the package to the monitor system used: the availability of manuals, training aids, primers and sample problems; the evolution and maintenance of the language and the package; and the establishment and operation of a users organization to exchange information about the language.

The choice of a language and package is not an exact process and because of the variety and range of evaluation criteria that are possible, it is doubtful whether it ever will be. This paper concentrates on the comparison of features of languages and of packages designed and/or used for simulation; user experience is mentioned as appropriate.

II. The Simulation Technique and Simulation Languages

1. Simulation as a Technique for Problem Solving and Research

Computer simulation has come into increasingly widespread use to study the behavior of systems whose state changes over time. [See, for example, IBM Corporation, Bibliography on Simulation (1966)]. Alternatives to the use of simulation are mathematical analysis, experimentation with either the actual system or a prototype of the actual system, or reliance upon experience and intuition. All, including simulation, have limitations. Mathematical analysis of complex systems is very often impossible; experimentation with actual or pilot systems is costly and time consuming, and the relevant variables are not always subject to control. Intuition and experience are often the only alternatives to computer simulation available but can be very inadequate.

Simulation problems are characterized by being mathematically intractable and having resisted solution by analytic methods. The problems usually involve many variables, many parameters, functions which are not well-behaved mathematically, and random variables. Thus, simulation is a technique of last resort. Yet, much effort is now devoted to "computer simulation" because it is a technique that gives answers in spite of its difficulties, costs and time required.

It is convenient to classify simulation models into two major types: continuous change models and discrete change models. Some problems are clearly best described by one type or the other; for some problems either type might be used.

2. Continuous-Change Models

Continuous-change models are appropriate when the analyst considers the system he is studying as consisting of a continuous flow of information or material counted in the aggregate rather than as individual items. These models are usually represented mathematically by differential or difference equations that describe rates of change of the variables over time. Such models have long been used in the physical sciences and engineering. Continuous-change models have also been applied to economics and the social sciences [see, for example, Tustin (1953), Allen (1956), and Beach (1957)]. They have been applied in various operations research problems [see, for example, Morgenthaler, (1961)].

If possible, the analyst uses analytical or numerical techniques to solve the system of differential or difference equations. If no available techniques are powerful enough or appropriate, the analyst might consider simulation. Such continuous-change models are naturally suited to electronic or mechanical analog computation. Unfortunately, analog machines often cannot be used because the functions involved are discontinuous, the number of variables are large, some of the variables involved are random variables, some of the variables must be non-negative or some of the operations required are not easily performed on analog machines.

Continuous-change models can be simulated on digital computers by using finite-difference equations which, in the limit, approach the differential equations of continuous flow. The problem definer using finite differences first selects, as the state variables, the factors of interest. Let $X_i(t)$ denote the value of the ith factor at time t and $X(t)$ the vector of all the X_i. $X(t)$ then becomes the state vector which describes the state of the system at time t. The problem definer next states the way in which the system changes over time. Symbolically, this may be represented by

$$X(t + \Delta t) = g(\overline{X(t)}, Z(t), W)$$

where $\overline{X(t)}$ represents the state vectors for all previous values of t,
$Z(t)$ represents the vector of values of exogenous variables for all relevant values of t,
W represents the vector of parameter values,
g specifies the behavior of the system.

The formulation of a continuous-change model consists of (i) identifying the state variables X, the exogenous variables Z, and the parameters W, and (ii) developing the functional relationship, g. The computational problem is to compute $X(t + \Delta t)$ when $X(t)$ is given. The result of the computations, that is, the output of the simulation, is a table or graph of $X(t)$ as a function of time or summaries of these data.

The "continuous change" world view has been extensively developed by Forrester [see *Industrial Dynamics* (1961), Ch. 6]. He conceptualizes the world as made up of money, men, material, information and capital goods, each of which flows from "level" to "level" with rate of flow controlled by "valves." A language, a flowcharting convention and a compiler, DYNAMO, have been developed to provide a means of using these ideas. The simulation advances in discrete time intervals of length DT and each state variable is computed as

$$X_i(t + \Delta t) = X_i(t) + (\Delta X_i)DT$$

where ΔX_i is an arithmetic expression. The effect is to compute values of the function $X_i(t)$ by the method of first differences. Experience indicates that the language is relatively easy to learn and that simple problems can be programmed and run in a relatively short time. Unfortunately the language has limitations for larger problems. Probably the most annoying limitation is that the basic language does not allow the use of subscripts. Different categories of inventory, for example, must be identified by different names and statements must be repeated for each category. Further restrictions are the use of fixed-time increments, computation of all statements at each time interval, first-order difference approximations to continuous functions, the limited number and rigid form of statements permitted, the restricted manner in which past data can be used, and the inability to enter exogenous data.

Continuous-change models can also be programmed in either general purpose languages or some of the discrete simulation languages. In certain cases, this is as easy and efficient as in DYNAMO, particularly if subscripts are required.

An efficient, widely used method for the solution of "continuous flow" problems is the use of hybrid analog-digital computers [see Scramsted (1962)]. There are languages developed for this application; see Table II.1. Brennan and Linebarger (1964) and (1965) and Clancy and Fineberg (1965) present comprehensive comparisons of these particular languages.

3. Discrete-Change Models

In discrete-change models, the changes in the state of the system are conceptualized as discrete rather than continuous. Systems are idealized as network flow systems and are characterized by the following:

—the system contains "components" (or "elements" or "subsystems") each of which performs definite and prescribed functions;
—items flow through the system, from one component to another, requiring the performance of a function at a component before the item can move on to the next component;
—components have finite capacity to process the items and therefore, items may have to wait in "waiting lines" or "queues" before reaching a particular component.

The main objective in studying such systems is to examine their behavior and to determine the "capacity" of the system: e.g., how many items will pass through the

Table II.1. General Characteristics of Selected Continuous-Change Simulation Languages for Digital Computers[a]

Name	*Meaning of Name*	*Availability Date*	*Originating Organization*	*Machine Implemented*
DEPI	Differential Equation Pseudo Interpreter	1957	Jet Propulsion Lab.	Burroughs 204
DEPI 4		1959	Allis Chalmers	IBM 704
DAS	Digital Analog Simulator	1963	Martin Co., Orlando, Florida	IBM 7090
DYSAC	Digitally Simulated Analog Computer	1961	U. of Wisconsin	Interpreter, in 1604 assembler language, for Wisconsin monitor
MIDAS	Modified Integration DAS (alias: much improved DAS)	1963	Wright-Patterson Air Force Base, Dayton, Ohio	In FORTRAN II for 7090 & 7094, in FORTRAN IV for 7094 & 7040, all interpreters
DIDAS9	Digital Differential Analyzer Simulator	1957	Lockheed, Georgia	IBM 704
DYNASAR	Dynamic Systems Analyzer	1962	Jet Engine Div. of General Electric	IBM 704/7090
PARTNER	Proof of Analog Results Through Numerically Equivalent Routines	1962	Aeronautical Div. of Honeywell	IBM 650, Honeywell H-800/1800
PACTOLUS	The river in which Midas got rid of the golden touch	1964	IBM Research, San Jose, California	IBM 1620
HYPLOC	Hybrid computer block-oriented compiler	1964	U. of Wisconsin	Compiler, in 7090/94 assembler language
FORBLOC	FORTRAN compiled block-oriented simulation language	1964	U. of Wisconsin	In FORTRAN
COBLOC	CODAP language block-oriented compiler	1964	U. of Wisconsin	Compiler, in 1604 assembler language
ASTRAL	Analog Schematic Translater to Algebraic language	1958	Convair	IBM 7094
JANIS		1963	Bell Telephone Laboratories	IBM 7090
DES-1	Differential Equation Solver	1963	Scientific Data Systems	SDS 9300
DYNAMO		1962	MIT, Industrial Dynamics Group	IBM 709/7090

[a] For a more complete list, see Clancy and Fineberg (1965)

system in a given period of time as a function of the structure of the system? The analytical techniques which may be used to solve such problems are queueing theory and stochastic processes. Examples of problems which have been formulated and studied as discrete change models are job shops, communication networks, logistics systems and traffic systems.

The computation in this type of simulation consists to a large extent in keeping track of where individual items are at any particular time, moving them from waiting line to component, timing the necessary processing or functional transformations and removing and transporting the items to other components or waiting lines. The result of a simulation "run" is a set of statistics describing the behavior of the simulated system during the run.

Special packages have been prepared for certain specific applications. To use these, the user supplies parameter values, data, and control values to adapt the program to his own model. Examples of such packages are the IBM-GE Job Shop Simulator and the Job Shop Simulator produced by Ginsberg, Markowitz and Oldfather (1965).

If such special simulators are not available or if the user decides not to use them, he can use general purpose languages such as FORTRAN, ALGOL, and PL/I. In the past, programs for the simulation of discrete-change systems on electronic digital computers were usually written directly in assembly languages or in general purpose language such as these.

But because simulation of discrete-change models does involve computations which are common to many models, a number of language and packages have been designed for formulating and writing a program for any discrete-change model. Table II.2 presents characteristics of some of these discrete-change packages: the name of the package; the general purpose language involved, if any; the originating organization; the machines for which the package is available and whether the package is currently available or planned. Six of these are discussed in more detail in Section III.

References to manuals and specifications of these languages and packages are given in Section V.3. A selected bibliography on the methodology of discrete-change simulation is given in Section V.2.

4. Evaluation of Languages for Simulation

The choices an analyst faces in attacking a problem are outlined in Table II.3. Obviously he must first choose the

Table II.3. Solution Techniques for the Two Types of Models

Solution Technique	*Type of Model*	
	Continuous change	*Discrete change*
Analytical technique	Difference equations Differential equations Calculus of variations Maximum principle	Queueing theory Stochastic processes
Simulation:		
(a) General purpose packages	Assemblers, FORTRAN, MAD, ALGOL, PL/I	
(b) Special purpose packages		IBM-GE Job Shop Simulator, etc.
(c) Simulation	See Table II.1	See Table II.2

Table II.2. General Characteristics of Various Contemporary Discrete-Change Simulation Packages

Simulation Package	*Computer Language*	*Originating Organization*	*Machines Implemented*	*Availability*
CLP	CORC	Cornell U.	CDC 1604	Current
CSL	FORTRAN	Esso, Ltd. & IBM U.K.	IBM 7090	Current
CSL 2	FORTRAN	IBM U.K.	IBM 7090/7094	In preparation
ESP	ALGOL	Elliott	Elliott 503 & 803	—
FORSIM IV	FORTRAN	MITRE	IBM 7030 & others with FORTRAN compilers	Current
GASP	FORTRAN	U.S. Steel Corp.	IBM 7040/7044, 7090/7090, 1620, 7070/7074, CDC G20	Current
GPSS	FAP	IBM	IBM 7090	Current
GPSS II	FAP	IBM	IBM 7090/94, IBM 7040/44	Current
	FORTRAN	UNIVAC	UNIVAC 1107	Current
GPSS III	MAP	IBM	IBM 7090/7094, IBM 7040/7044	Current
	—		IBM S/360	In preparation
GSP 2	—	U.S. Steel Co. Ltd.	Ferranti Pegasus	Current
			Elliott 503	In preparation
Job Shop Simulator	—	IBM-GE	IBM 7090	Current
MILITRAN	—	Systems Research Group for Office of Naval Research	IBM 7090/7094	Current
OPS	—	MIT	IBM 7090/7094	Current
QUICKSCRIPT	—	Carnegie Inst. of Technology	CDC G20	Current
SILLY	—	U.S. Steel Corp.	—	In preparation
SIMON	ALGOL	Bristol College of Sci. & Tech.	Elliott 503 & 803	—
SIMPAC	SCAT	Systems Development Corp.	IBM 7090	Not available
SIMSCRIPT	FORTRAN	RAND (SHARE)	IBM 7090/7094, 7040/44	Current
	FORTRAN	California Analysis Corp.	IBM 7090, 7090/7094, 7040/7044 CDC3600, 3800, 6400, 6600, 6800 Philco 210, 211, 212; UNIVAC 490, 1107, 1108	Current
	—	Digitek	GE 625/635	Current
SIMTRAN	FORTRAN	MITRE	IBM 7030	Current
SIMULA	ALGOL	Norwegian Computing Center	UNIVAC 1107	In preparation
SOL	ALGOL	Burroughs, Case Inst. of Tech.	Burroughs B5000/5500	—
			UNIVAC 1107	Current
UNISIM	FAP	Bell Labs.	IBM 7090/7094	Current

type of model; in some cases, of course, the problem clearly indicates which one should be chosen. It is true that there are many problems that are best considered in terms of discrete changes, and it makes little sense to try to force these into the continuous-change model. However, there are many other problems which can be investigated efficiently by continuous-change models. These include cases where levels of operation actually change continuously, such as in pipeline networks, as well as cases where items are discrete but may be considered in the aggregate and measured continuously. The limitations of the DYNAMO package should not necessarily be considered as limitations of the continuous-change approach to model formulation.

Once this choice has been made, the analyst must pick a solution technique. If he selects simulation, he has two and perhaps three choices available. He may select a general purpose package (an assembly language, FORTRAN, ALGOL, etc.) or a simulation language or package, or he may find or write a program specially designed for his problem. It is, of course, possible to write simulation programs using general purpose assemblers and compilers. A survey conducted by Chen (1964) confirms the obvious conclusion that in the past most such programs have been so written. Indeed, some believe that general purpose languages are almost always preferable, providing efficiency without important disadvantages [see, for example, Fedderson and O'Grady (1965)]. Furthermore, available evidence supports another obvious conclusion: programming in lower language systems such as assemblers takes more programming time but results in object programs with less execution time than programming in higher level systems. Hence, one might do the simulation first in a higher level language; if execution time is excessive, the analyst or his programmer can reprogram in a lower level system the parts that are taking the most time.

III. Comparison of Six Discrete Flow Simulation Packages

1. Descriptions of Six Languages

New simulation packages continue to be developed, motivated both by the introduction of new machines and by dissatisfaction with existing packages. It seems evident that simulation languages, for many applications, have sufficient advantages over more general purpose languages to justify their use. Indeed, the substantial number of simulation languages for discrete-change models now available has introduced a new dimension to computer simulation. A potential user can choose from among several competing packages, each of which offers some features suitable for his particular problem or class of problems.

The purpose of this section is to compare six discrete-flow simulation languages and packages in detail. The six and the reasons for selecting them are: SIMSCRIPT because of its unique features; GASP because it is a set of subroutines written in a widely used general-purpose language, FORTRAN; GPSS because it is probably the most used and is a package provided and maintained by the largest manufacturer of electronic computers; CORC (with CLP) because it is part of a package designed specifically for teaching; CSL because it differs from the others in the way that it controls the simulation and because it was developed outside the United States; and SOL because it is based on ALGOL.

References to detailed specifications of all the packages and languages listed in Table II.2 are given in Section V.3. A brief description of the objectives of the designers and implementers for the six selected languages is given below because it is reasonable to discuss the characteristics of a language (package) in terms of the objectives of the designer (implementer).

GPSS [IBM, Reference Manual]:

"The program described herein is a general-purpose simulator designed to aid system study work. The system to be simulated must be described by the user in terms of a special block diagram. The program operates on the IBM 7090 under the IBSYS/FORTRAN System, *and no knowledge of the computer operation is assumed.* The user need only know the rules by which the block diagrams are constructed.

"The simulator allows the user to study the logical structure of the system. The flow of traffic through the system may be followed, and the effects of competition for equipment in the system may also be measured. Computer output may be arranged to provide information on (1) the volume of traffic flowing through sections of the system, (2) the distribution of transit times for the traffic flowing between selected points in the system, (3) the average utilization of elements in the system, and (4) the maximum and average queue lengths at selected points in the system.

"Various statistical and sampling techniques may be introduced into a GPSS II model. Levels of priority may be assigned to selected units of traffic, and complex logical decisions may be made during the simulation. It is also possible to simulate the interdependence of variables in the system, such as queue lengths, input rates and processing time."

CORC and CLP [Conway, Maxwell, and Walker]:

"Programming is the process of describing a computational task in a form and a language that will be intelligible to a computer. Although there are several different types of languages that can be employed, certainly the easiest to use from the point of view of the person doing the programming is one which is not unlike familiar mathematical notation and which uses English words in a reasonably formal manner. FORTRAN and ALGOL are the two most widely used languages of this type. Because they are meant for professional use in a wide variety of problems they have considerable complexity and take a fair amount of time to master. For quick learning for the beginner a less sophisticated language is desirable, even at the expense of some grammatical clumsiness and some limitation in the variety of problems it can handle. CORC is such a language."

GASP [Kiviat]:

"GASP is a FORTRAN-compiled, simulation-oriented programming language. Simulation models, when expressed in GASP-oriented flow-charts, are easily transcribed into machine-executable FORTRAN statements. . . . Simulation models are usually designed by operating or engineering personnel who are unfamiliar with computer programming; they are coded and debugged by com-

puter people equally unfamiliar with the processes being simulated. GASP is intended to bridge the gap between these two groups. Engineers can formulate their problems in a machine-independent language; programmers, familiar with the language, can easily adapt their thinking to the problem and code the model. GASP also provides debugging features that expedite the testing and validation of the model. . . .

"GASP views simulation problems in a highly normalized manner. Precise definitions are established for isolating different parts of systems, for naming these parts and for specifying possible relations between them. This specificity of structure allows GASP programmers to construct simulation models of systems rapidly and economically. It provides a programming compatability that allows programs of simulation models to be united into larger, more comprehensive programs with little or no change. It provides a standard of performance by which programming time and program execution may be evaluated. It provides a machine-independent and easily expandable programming language. And perhaps most important, it is simple, straightforward, and easy to learn."

SIMSCRIPT [Markowitz, Hausner, and Karr]:

"The SIMSCRIPT system described in this manual was developed to meet the need to reduce programming time. It also provides increased flexibility in modifying such models in accordance with the findings of preliminary analysis and other circumstances. . . .

"Fortunately, experience now confirms that much of the time spent in both logical formulation and actual programming is spent on operations that are often similar from one simulation problem to the next. Thus there is a clear opportunity and need for a programming system specially adapted to the problems of writing simulation programs. SIMSCRIPT was designed to answer this need. . . .

"Any digital simulation consists of a numerical description of the "status" of the simulated system. This status is modified at various points in simulated time which may be called "events." SIMSCRIPT provides a standardized definition-form for specifying the status description. It also automatically provides a main timing routine to keep track of simulated time and the occurrence of events. An "event routine" is then written for each kind of event, describing how the status is to change. The SIMSCRIPT source language is specifically designed to facilitate the formulation and programming of these event routines.

"Although SIMSCRIPT was developed for simulation problems, and the present exposition is presented in terms of simulation problems, SIMSCRIPT is actually a general programming system that is also readily usable for non-simulation problems."

CSL [IBM United Kingdom, Ltd, and Esso Petroleum Co. Ltd]:

"Control and Simulation Language (CSL) is designed for the formulation, as computer programs, of the complex decision-making problems which arise in the control of industrial and commercial undertakings.

"In the field of simulation, CSL eases the construction and expression of a logical model to represent the system under study and provides built-in facilities for running the model on the computer.

"The language is based on the use of groups of entities which are the elements of the system, and in particular on subgroups (or sets) of entities which have some common property. The majority of CSL statements make use of sets by examining their membership, or by operating in some way on a complete set, or on a selected member."

2. PREVIOUS COMPARISONS

Comparisons of discrete change simulation packages have been made by Krasnow and Merikallio (1964), Freeman (1964), Murphy (1964), Young (1963), Ginsberg (1965), and Tocher (1965). The papers by Krasnow and Merikallio, Freeman and Tocher are published in generally available journals.

Murphy reports on a computer simulation model programmed in both GPSS II and SIMSCRIPT. The two programs produced identical output from the same input after appropriate adjustments were made in random number generation, the order in which events are caused, the treatment of simultaneous events, and differences in number rounding. Murphy recommends SIMSCRIPT in preference to GPSS II because SIMSCRIPT resulted in less execution time (3.6 minutes for SIMSCRIPT compared to 26.8 minutes for GPSS in a particular case on the IBM 7090) and requires less memory (10,000 vs. 20,000 words) and because of the following features:

(i) SIMSCRIPT can operate with variable-length time increments but GPSS II cannot;

(ii) The SIMSCRIPT Report Generator is very convenient;

(iii) Work packing is available in SIMSCRIPT but not in GPSS;

(iv) In SIMSCRIPT the size of arrays need not be known at compilation time; and

(v) In SIMSCRIPT but not in GPSS II variable names can be assigned to parameters for ease of identification.

Note that the comparison involved GPSS II. Some of the limitations of GPSS II that Murphy reports would not apply to GPSS III.

Young reports on her experiences with SIMPAC, SIMSCRIPT and GPSS. She lists conditions under which each of these might be preferred, preferences that are determined primarily on the basis of subjective evaluations.

Ginsberg limits his comments to GPSS and SIMSCRIPT ". . . because they received the most interest amongst the existing languages." He states his conclusions as follows:

"In summary, we would answer the crucial question as to which language to use for a given simulation model by stating: if it is possible to write the program in GPSS, if memory limitations will not be exceeded, and if the larger running time is not 'excessive' then GPSS should be used. Otherwise SIMSCRIPT or one of the other languages should be used. Obviously, these judgments are highly subjective but must and can be made before undertaking all but very small simulation experiments. In order to make these judgments, there must exist one person who has a fairly complete understanding of the proposed model, of the experiments to be performed, and of both languages. This kind of person is by no means easy to come by, but is necessary to insure any kind of rational decision in the language selection problem."

3. COMPARISON OF SIX SIMULATION PACKAGES

Simulation languages differ from general purpose programming packages in that simulation languages include features which improve the communications between problem definer and programmer and simplify the programming of the computations that are characteristic of simulation problems. These ends are accomplished primarily through the following five capabilities:

—the capability to impose a fixed "structure" on the

assignment of computer memory to variables and data. This assignment is more complete, detailed, and specific than that used in most general purpose languages.

—commands to facilitate changing the state of the sjmulated world. In most cases this is done by a "master" or "timing" routine that controls the sequence in which subprograms are executed

—commands which facilitate the determination of whether or not a subprogram is to be executed at a particular time.

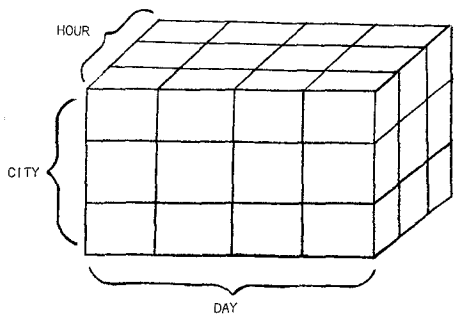
—commands to facilitate computations that are used frequently, in particular those dealing with random numbers and probability distributions.

—commands which facilitate the recording of statistics during program execution and the reporting of results after the simulation run is ended.

Simulation languages differ in the means used to provide these five capabilities. A comparison, in tabular form, is given for six languages. Tables III.1–III.5 also include a comparison of some characteristics, not necessarily unique to simulation languages, that are important and useful.

(a) *Structure of Memory Assignments—Data Structures.* Most users are not concerned with the details of program compilation and execution. Nevertheless, the comparison of languages can be aided by examining details such as the structure of memory assignments. In simulation, as in all computer programs, some memory cells are used to store data and others to store instructions; the contents of some of these cells are changed as a result of the execution of the instruction. In languages such as FORTRAN and ALGOL and in general purpose assemblers, the programmer is free, in the source program, to define variables as he wishes with minor restrictions such as distinguishing between integers and floating-point numbers. In the resulting object program, the compiler or assembler assigns memory locations to each variable as required. Essentially, therefore, most compilers, interpreters, and assemblers treat all variables in the same way. In simulation languages, on the other hand, several types of variables usually can be defined.

Variables in languages such as FORTRAN and ALGOL may be subscripted (be defined as "arrays"). For example, suppose a variable of interest in a problem is temperature, called TEMP. The value of this variable may depend on city, day and hour. The possible values of the variable, TEMP, might be stored in a three-dimensional array as follows:



Any particular value could be denoted by specifying the values of the arguments in an identifier such as "TEMP (CITY, DAY, HOUR)." This particular way of storing the data and identifying datum is, of course, not unique. The three-dimensional array, for example, could be broken up into a number of two-dimensional arrays or even one-dimension arrays. Obviously too, the order of the subscripts is arbitrary.

Most computer simulation languages have the capability of declaring and using subscripted variables just as in the general purpose languages. Several of the simulation languages provide for additional capability in storing data and identifying them in the source language. The reason is that there are some variables in almost every simulation model which are treated in special ways so that it is worthwhile to provide specific mechanisms and specific names in the language and in the package for them. In the discussion which follows, names such as record, field and list are used to distinguish these variables from "ordinary" variables. It is emphasized that logically these are "variables," no different than any other variables.

The name used in this paper for an item of data in a simulation language is "RECORD." A record is a one-dimensional variable consisting of one or more fields of data. A "FIELD" is a basic unit of data and may consist of one or more binary digits, decimal digits, alphabetic, or alphanumeric characters. A field may be stored either in a cell, a part of a cell, or in several cells of the computer.

The basic use of a record is to describe an object in the simulated world. The data in the fields of a record describe the "properties" of the object. The record and its fields are usually given names or identifications; it is conventional to use names which are mnemonics for the object being represented.

To illustrate these definitions, consider the simulation of a barber shop. "Barbers" and "customers" might be objects being simulated. An individual barber might have the following properties or characteristics:

his name
the time he begins his shift
the time he completes his shift
average time to complete a haircut
whether idle or busy at a particular instant of time

The data describing this barber may appear as follows:

Name	Time In	Time Out	Average Time	Status
Jones, R.	0800	1600	010	Busy

There would probably be several barbers, and there would be a record to describe each one. It is convenient to have a name for a set or group of records describing similar objects, in this case, barbers. The group of records might be called BARBS and would appear as follows:

Name	Time In	Time Out	Average Time	Status
Jones, R.	0800	1600	010	Busy
≈				≈
Smith, J.	0900	1700	009	Idle

Actually, this "GROUP OF RECORDS" is nothing more than a two-dimensional array. The designers of many simulation languages found it useful to provide ways of locating a particular item of data by conventions different than those used in most general purpose languages. In our illustration, a property of a particular barber is often of interest. Instead of identifying these properties by BARBS (NAME, PROPERTY), a simulation language might provide the capability of identifying it by NAME (BARBS) or PROPERTY (BARBS) on the justification that such a procedure is more "natural" and meaningful for simulation modeling.

Another capability that is often found in simulation can be illustrated by the property "status." During the course of the simulation, this property changes. When a barber finishes a haircut, the status in his record must be changed to "idle" if no customer is waiting. When a customer arrives, the program must examine the status of each barber to determine those who are idle. It may be better if a separate list were kept of all barbers who are idle. Then, when any change occurs, only the list need be updated. Some simulation languages provide special capabilities for maintaining such "status" lists, that is, lists of names or identifiers of records ("LIST OF RECORD NAMES") that have certain properties.

The advantages of such a capability are: the procedure saves memory space because the property need be stored only when appropriate; the procedure saves computer time because all possible records do not have to be searched each time there is a change in the state of the simulated world; problem definition and programming time are reduced because such lists are a natural mode of expressing the state of the simulated world.

Another capability of simulation packages is designed primarily to use memory space efficiently. This may be illustrated in our example by the records for customers. A group of records for customers might appear as follows:

CUSTOMER RECORDS

Name	Time A	Time B	Time F	Type of Service
CUST 1	617	648	723	CC
≈				≈
CUST 200				

Text is continued on page 734

TABLE III. 1. STRUCTURE OF MEMORY ASSIGNMENTS-DATA STRUCTURES

	GPSS II[a]	*SIMSCRIPT*	*CLP*	*CSL*	*GASP*	*SOL*
1. Object being simulated: fundamental element (Record)	'Transaction'[b] 'Storage' 'Facility' 'Queue'	Individual entity	'Entity'	'Entity'	'Element'	Variable, local & global 'Transaction' 'Facility' 'Store'
2. Properties of objects (Fields)	Transaction: 'Parameter' 'Priority' 'Mark Time' Storage: 'Storage Capacity' 'Maximum Contents' 'Current Contents' Utilization time integral Total entries Facility: 'Status' Utilization integral Total entries Queue: 'Maximum Contents' 'Current Contents' Utilization time integral Total entries	'Attribute'	'Attribute'	'Array'	'Attribute'	Value
3. Group of objects being simulated (Group of records)	'Transactions' 'Storage' 'Facilities'	'Entity'	'Class'	'Class'	'Element List Matrix'	Subscripted variable
4. Data about the environment (Variables)	'System Variable' 'Savex' 'Function' 'Frequency Table' [*GPSS III* 'System Numerical Attributes (SNA)']	'Permanent System Variable' ('Array Number')	System variables	'Array'	System variables	Global variables & tables
5. List of names of objects having certain properties (List of record names)	'Events Chain' 'Assembly Set' 'Service Chains' 'Interrupt Chains' (Delayed Transactions) [*GPSS III* 'User Chain']	'Set'	'List'	'Set'	'Element List' or 'Queue'	
6. Can records be temporary?	Yes	Yes	Yes	No	Yes	Yes

[a] Names in quotes are actual names used in language indicated.

[b] Entries under GPSS are for GPSS II; GPSS III changes are listed separately if a change has been made. Unless specifically noted, GPSS II features are retained in GPSS III.

TABLE III. 2 CHANGING THE STATE OF THE SIMULATED WORLD

	GPSS II	*SIMSCRIPT*	*CLP*	*CSL*	*GASP*	*SOL*
1. Subprogram: agent of change	Block subroutines: specific blocks for 36 basic types of system action	'Event' subroutine	Block; Subroutine	'Activity' subroutine	'Event' subprogram causing 'Activity'	'Process' statement; procedure
2. Who provides subprogram?	GPSS (except 'Help' provided by user)	User	User	User	User	User
3. Time control routine	Main scanning routine [*GPSS III* GPSS III main scanning routine]	'Timing Routine'	Programmed by user	Timing routine	'GASP Executive'	Programmed by user, using 'Wait' statement
4. Amount of time advance	To next scheduled 'Future Event' after completion of all possible 'Current Events'	To next imminent event	Programmed by user	To next imminent event, by minimum value in T-Cell associated with each entity	To next scheduled event	To next point of program after a Wait Statement is completed
5. Exit after time advance	To appropriate block subroutine	Control transferred to appropriate event subroutine	Programmed by user	All activity subprograms are activated cyclically	Control transferred to appropriate event subroutine	To active program
6. Return of control to timing routine	By block subroutine	By event subroutine	Programmed by user	Automatically, when no activity can be executed	By event subroutine	Programmed by user
7. What flows in simulated world?	'Transaction'	'Temporary Entity'	Programmed by user	'Entity'	Temporary 'Element'	'Transaction'
8. What determines when change occurs?	System status; change of status forces new events; scan before time advance	Tests in event subroutines	Programmed by user	Tests in activity subroutine	Tests in event subprogram	Determined by user Parallelism by a 'duplicate' operation
9. Can changes be caused externally?	'Jobtape' for exogenous events 'Help' for arbitrary modification	Yes: Exogenous event tape			Yes; load any number of exogenous events as data input	Yes

TABLE III. 3. COMMANDS TO FACILITATE SUBPROGRAM EXECUTION

	GPSS II	*SIMSCRIPT*	*CLP*	*CSL*	*GASP*	*SOL*
1. Create temporary records	'Originate' 'Generate'	'Create'	'Let'	No temporary entities	Temporary elements are created by naming, are stored in queues, may cease to exist upon departure from last queue	Declaration Schmoo Process (reproduction) 'Start Statement' Input statements
2. Remove temporary records	'Terminate'	'Destroy'	'Erase'			'Cancel'
3. Place (or remove) event on schedule	Main scanning routine is accessible by 'Priority,' 'Buffer,' 'Advance,' 'Help'	'Cause' 'Cancel' Exogenous event	Programmed by user	T-Cell for each entity gives time available	'Schdl' 'Remove'	Automatic
4. Change list membership	'Seize' 'Release' 'Interrupt' 'Hold' 'Preempt' 'Leave' 'Return' 'Enter' 'Store' 'Gate' 'Link' 'Unlink' 'Queue' [*GPSS III* 'Depart' 'Test' no 'Hold' no 'Store']	'File' 'Remove' specific item 'Remove First'	'Insert' 'Remove'	'Load' 'Gains' 'Zero' 'Loses' 'Converse'	'Filem' 'Fetchm'	'Seize' 'Release' 'Enter' 'Leave' 'Wait' 'Wait Until'
5. Sequencing in list	'Current' events chain by priority by delay by FIFO; 'Future' events chain by departure time by FIFO; 'Service' by priority by FIFO	FIFO LIFO Ranked on attribute value	'First' 'Last' Ranked on attributes Removal by specifying entity identification	Add to 'Head' or 'Tail' of Set; 'Rank' set members on specified criterion	FIFO LIFO High or low ranking of attribute value	Priority 'Control Strength' First request
6. Logical commands and phrases	Selection modes: Both All Pick P FN SIM blank Gate conditions: NU SE SNF M U SNE LS NM I SF LR Algebraic 'Compare' [*GPSS III* Selection Mode: SBR]	'For Each' 'Loop' 'Find Max' 'Repeat' 'Find Min' 'Or' 'Find First' 'And' 'Where' 'If' 'With' 'If Empty' 'Go To'	'If' 'Repeat' Simplified sequence control available to allow automatic execution of next statement in program if tested condition is false. Multi-way branching available with single 'if' statement.	'Chain' 'For' 'In' 'Not In' 'Equals'	FORTRAN	ALGOL

TABLE III.4 PROGRAMMING FEATURES

	GPSS II	*SIMSCRIPT*	*CLP*	*CSL*	*GASP*	*SOL*
1. Basic unit of program	'Block'	SIMSCRIPT statement Event routine	CLP or CORC statement Block Subroutine	CSL statement Activity routine	FORTRAN statement Event routine	SOL statement
2. Programming requirements	None other than GPSS (except 'Help' in FAP) [*GPSS III* (MAP)]	Must know FORTRAN	Must know CORC	FORTRAN knowledge useful	Must know FORTRAN	ALGOL knowledge useful
3. Does language provide specific flowchart symbolism for program expression?	Yes	No	No	No	Yes	No
4. Recursion: infinite nesting	No	No	No	Yes, on logical test chains	No	Yes
5. Arithmetic (data-changing) commands	'Assign' 'Help' 'Tabulate' 'Savex' ('Variable Statements') [*GPSS III* 'Savevalue' no 'Savex']	'Let' 'Store' 'Compute' 'Do To' (FORTRAN statements)	CORC statements	FORTRAN statements	FORTRAN statements	ALGOL statements
6. Commands to collect statistics	'Tabulate' 'Queue' 'Savex' 'Hold' 'Help' 'Store' 'Seize' 'Release' 'Enter' 'Leave' [*GPSS III* 'Depart' no 'Store' no 'Hold']	'Accumulate' 'Compute' Number Sum- Sum Squares Mean Mean- Square Variance Standard Deviation	CORC statements	'Hist' 'Sum'	'Collect' 'Histog'	'Tabulate' statement
7. Functions, distributions, random numbers	'Functions': argument any standard system variable Uniformly distributed random numbers [*GPSS III* Any SNA can be a dependent variable of a function]	Uniformly distributed random numbers Non-unifom continuous or discrete probability distributions	Uniform random distribution Exponential distribution CORC functions (square root, arctan, max. min etc.)	Multiple random number streams to facilitate introduction of independent random variables: Normal distribution Rectangular Negative exponential Arbitrary	Option-random operation or random decision Erlang distribution Normal Poisson Uniform Random numbers from probability list Regression equation	Rectangular distribution Exponential Poisson Normal Geometric Arbitrary
8. Input-output	Built-in fixed I/O: 'Saves' transfers model to tape 'Reads' restores model from tape 'Write' places transactions on tape 'Jobtape' recovers transactions from tape	Input-output commands: 'Save' 'Endfile' 'Read' 'Load' 'Read From' 'Record Memory' 'Write On' 'Restore Status' 'Advance' 'Backspace' 'Rewind'	CORC I/O CLP tape read and write	FORTRAN input-output statements 'Output'	Subroutines 'Datain' and 'Output' GASP summary report 'End Run' (optional) Stacking of runs, data decks in sequence	Cards & tape 'Read' 'Write'
9. Report output	'Print' (Savexes) Normal output: Model listing Clock time Block counts Savexes Facility statistics Storage statistics Queue statistics Frequency tables Summary statistics Error Conditions 'Help' (arbitrary reports) [*GPSS III* can print any equipment statistics]	Report generator	Report writer CORC output statements	FORTRAN output statements	'GASP Summary': contents of all queues, max. and ave. queue length, scheduled but unexecuted events 'End of Run': written by user for output beyond that of 'Summary'	Output statements Many automatic summaries Debug capability
10. Use for non-simulation	No	A general purpose language User-supplied 'Main' control routine replaces timing routine	CORC language is a proper subset of CLP	No	GASP is imbedded in FORTRAN; subroutines can be used for any purpose	Not intended, but possible

TABLE III.5. MECHANICS OF USE

	GPSS II	*SIMSCRIPT*	*CLP*	*CSL*	*GASP*	*SOL*
1. Implementation:	IBM Corp.	RAND Corp.	Cornell University	IBM-UK	P. Kiviat, RAND Corp.	Case Institute of Tech.
Implementer computers	IBM 7090/94, 7040/44	SHARE: IBM 7090/94, 7040/44	CDC 1604	IBM 7090 (requires 1401-4 tapes for FORTRAN source statement conversion)	IBM 7040/44, 7090/94	UNIVAC 1107
Documentation	IBM Corp. Introductory manual Users manual Systems manual	Prentice-Hall, Inc.	Cornell	IBM-UK	P. Kiviat	Case
Training	IBM Corp.	California Analysis Center, Inc. Southern Simulation Service			None	
2. Compilation and running procedure	FAP Model deck is input to interpretive routines. Model deck may be altered by overlaying any model element. Models may be batch run. Block numbers can be supplied at input time. Run control cards: 'Jobtape' 'Job' 'Clear' 'Reset' 'Saves' 'Reads' 'Start' [*GPSS III* MAP Block numbers assigned by assembler.]	Compilation and execution by FORTRAN compiler SIMSCRIPT source programs translated into FORTRAN which are then compiled (Recent implementations compile directly into machine language) Exogenous event tape Record memory, restore status commands	Compile, load, and go	IBM 1401 prepares a FORTRAN II program for compilation and execution on IBM 7090	GASP Executive and source program compiled as FORTRAN program using standard FORTRAN compiler Batch processing of sequence of jobs. GASP Executive and source programs must be recompiled if max number of length of queues are changed: otherwise only 'Datain' and source program need be.	First pass by SOL compiler to interpretive pseudo-code, then through interpreter for execution.
3. Debugging and diagnostics	Dynamic error indications terminate run and cause printout of system status and accumulated statistics for source language debugging. 'Trace' allows transaction moves to be followed. Limited syntactical error checking at input.	FORTRAN diagnostics Report generator for snapshots	Extensive CORC diagnostics Syntax errors correction Program always compiles	'Check' FORTRAN diagnostics	FORTRAN diagnostics 'Monitor' program (optional) with 'Error' routine; trace prints event times; COMMON memory dump option	Debugging incorporated in model; selective tracing ALGOL run time diagnostics
4. Memory						
When dimensioned	At FAP assembly of GPSS [*GPSS III* At load time (MAP)]	Load time	In source program	In source program	In source program	As in ALGOL
Packing	Set by FAP program [*GPSS III* MAP]	Yes, up to 4 attributes per word	No	No	No	
Allocation	Fixed by FAP/MAP assembly Dynamic for transactions	Dynamic for temporary records	Dynamic for temporary records	Fixed	As in FORTRAN	Dynamic
External memory options	Tape	Tape	Tape	Tape	Tape (FORTRAN capability)	Tape
5. Speed Compilation Execution						
6. Experimentation	Models are self-initializing. 'Jobtape' can introduce an initial transaction load. Separate 'Prerun' may also be used to introduce load.	Initialization cards: values of all permanent entities and size of each entity (providing flexibility for changes without recompiling); previously compiled sections of programs need not be recompiled	Initial values in dictionary	Initializing routines	Programmed in 'Datain' or initializing data cards for each run in batch; control words to initialize some storage areas.	Initialized to zero, not busy, empty at creation, simulator in 'choice state'
7. Other implementations	UNIVAC 1107 (written in FORTRAN) IBM S/ 360, GPSS III only	California Analysis Corp: IBM 709, 7090/94 CDC 3600, 6800, 6400, 6600, 6800 Philco 210, 211, 212 UNIVAC 490, 1107, 1108 Digitek: GE 625/635		IBM Data Center, London, CSL2	IBM 1620 IBM 7070/74 CDC G-20	Burroughs B 5000/5500 (not released)

When a customer arrives, his name, time of arrival (Time A), and type of service required are recorded. The time of beginning of his haircut, Time B, and the time of finishing, Time F, are recorded when they occur.

During a simulation a large number of customers may be processed. The detailed data concerning each one, which must be available when each is in the system, is not of interest after a customer leaves. Only summary information about customers is needed for analysis. Therefore, it could be very useful to provide a capability of assigning memory space to a customer when he arrives and to erase the record when he leaves. This capability is provided in some languages by permitting the specification of "temporary" records.

The names attached by the analyst to "fields," "records," "groups," "variables" and "lists" in a source program are usually those of physical objects in the system being simulated. The following is an example of names that might be used in a barbershop simulation.

Physical object in simulation	*Represented in memory by*
All barbers	A group of records (called, say, BARBS)
All customers	A group of records (called, say, CUSTS)
A particular barber	One record (in the group called BARBS)
A particular customer	One record (in the group called CUSTS)
The properties of a barber, e.g., average time for a haircut	Fields (of the records called BARBS)
The properties of a customer, e.g., type of service required	Fields (of the records called CUSTS)
The idle barbers	List of names of those BARBS records which represent barbers who are idle
The customers waiting	List of names of those CUSTS records which represent customers who are waiting
The hours the barber shop is open	Variables TOPEN and TCLOSE

The six simulation languages analyzed here differ from general purpose languages in that they permit the analyst to specify more than one type of memory assignment by specifying more than one type of variable. The correspondence between the data about the simulated world, the way it is treated by the computer, and the names used in the various languages is as follows:

Data about the simulated world	*Computer assignment*	*Table III.1*
Objects being simulated	records	line 1
Properties of objects being simulated	fields	line 2
Groups of objects being simulated	groups of records	line 3
Other data about the simulated world	one-, two-, three-,. . . dimensional variables	line 4
Lists of objects having certain properties	lists of record names	line 5

Table III.1, in lines 1, 2, 3, 4 and 5, gives the actual names used in a language using quotation marks. For example, a group of objects being simulated is called a 'Class' in CLP and CSL, an 'Element List Matrix' in GASP; 'Transactions,' 'Storages' and 'Facilities' in GPSS; and 'Entity' in SIMSCRIPT. In SOL, any subscripted variable may be used for this purpose. Line 6 in Table III.1 shows that all languages except CSL permit the definition of temporary records.

There appears to be general agreement that the distinction between objects being simulated, properties of these objects, groups of these objects, data describing the environment, and lists of objects having a particular property is useful in formulating a simulation model and should be retained in future discrete-change simulation languages. However, the mechanics of this definition can become burdensome. There is also general agreement that provision must be made for temporary objects in order to make efficient use of memory space.

(*b*) *Changing the State of the Simulated World.* Computer programs are usually divided into parts, each of which is relatively small and each of which is logically distinct. The program is not only easier to write in this way, but changes or corrections can be made in one part without affecting others. The computer program is executed by providing means for determining the sequence in which the parts or subprograms are to be executed.

In computer simulation, a division of the program into subprograms representing "events" or "activities" in the simulated world has been adopted by most language designers as both natural and useful.

EVENT—Represents a change in the simulated world. The event is usually thought of as occurring instantaneously and taking no time, e.g., a barber finishes a haircut.

ACTIVITY—Represents an occurrence in the simulated world which takes time, e.g., a haircut.

In general any activity can be represented by two events, the event which is the beginning of the activity and the event which is the end of the activity. One can look at the simulated world as one involving either "events" or "activities." One can think of activities causing changes in the state of the simulated world which then creates events, or one can think of events as recording or marking instantaneous changes in the state of the simulated world and thus providing the means by which the duration of an activity is computed. It is possible to have circumstances in which an event can occur without a corresponding activity; for example, a barber may finish a haircut but cannot begin another one because there is no customer waiting. (However, he does then begin the activity of being idle.)

An activity or an event can only occur in the simulated world if certain conditions are satisfied. For example, a haircut can occur only if a barber is available and if a customer is waiting. The logical relationships of conditions in most simulations tends to be quite complex. In practice, one of the major difficulties is to state conditions within the simulation program so that the correct sequence of events and activities will occur during execution of the program.

Three different methods for the determination of the proper sequencing are used in the languages under consideration here:

—use of an Executive Routine that keeps track of "time" by a Schedule of Events and executes subprograms at the appropriate times (SIMSCRIPT, GASP, GPSS),

—use of an Executive Routine that keeps track of "time" without a Schedule of Events and executes subprograms in the order in which they appear in the particular program (CSL),

—use of no preprogrammed routine, leaving to the user the programming of sequence control (CLP).

In simulation packages which have an Executive Routine, a master or timing routine is employed to keep track of "time." Some simulation languages explicitly recognize a "schedule" or calender of events or activities. This is done by having the master routine transfer control to the subprograms at the appropriate times during execution of the program. The subprogram is then executed, and control is transferred either to another subprogram or back to the master routine. Simulation languages differ in the method by which the subprogram to be executed is selected and whether or not the exit from the subprogram must return to the master routine or passes directly to another subprogram. Such languages must have commands to place events on, or delete events from, the schedule at appropriate times. The particular subprogram being executed contains instructions representing the activity that follows. It also contains instructions that will determine what other events will occur in the future and when they will occur.

A language that does not explicitly recognize a schedule of events (or activities) obviously does not require commands to schedule or delete events. In such systems, each subprogram must be preceded by a sequence of tests which determine whether the particular subprogram can and should be executed (whether the activity can be performed) at a particular time.

A comparison of the six selected languages with respect to changing the state of the simulated world is shown in Table III.2. Each uses a subprogram as the basic building block of the program (line 1): In SIMSCRIPT, GASP and GPSS, the subprograms describe events, while in CSL they describe activities. In CLP and SOL, the user can choose either orientation. These are programmed by the user, except in GPSS in which 36 specific, standard subprograms are provided and in SOL and GASP in which some preprogrammed routines are also available, (line 2).

All packages, except CLP, provide an Executive Routine which, in addition to other functions, keeps track of time and determines the order in which subprograms are executed (line 3). In all, time is advanced to the next imminent event instead of moving forward through time in fixed increments (line 4). In CSL, the Executive Routine cycles through all subprograms, in order, until no more activities can occur; control is then returned to the Executive. In SIMSCRIPT, GASP and GPSS, control is transferred to the particular subprogram needed at the time and then back to the Executive (SIMSCRIPT and GPSS) or, at the option of the user, either to the Executive or another subprogram (GASP) (lines 5 and 6). It is not evident which of the two methods of control, the "schedule of events" or the testing for possible execution of all activities, is preferable [Laski (1965) and Tocher (1965)].

The names assigned to objects that "flow through" the simulated world in the several languages are listed in line 7 of Table III.2. Line 8 gives the conditions that determine when change occurs. SOL is the only one that provides explicitly for parallel (simultaneous) activities.

It is necessary to provide for events and activities created by factors outside the simulated world itself that could not be caused by events internal to the simulation. In simulating such situations, it is useful to have means to incorporate such "exogenous events" with ease into the subprogram-sequencing process. The availability of this feature is shown in Table III.2, line 9.

(*c*) *Commands to Facilitate Subprogram Execution.* Simulation languages have been designed with commands to make easy the order in which subprograms are executed. Table III.3 lists the names of these commands in the six language packages considered here. The first two lines give the commands that can be used to create "temporary" records (objects being simulated)—operations which are particularly important in discrete-system simulations. Line 3 gives the names of the commands used to place or remove events from the schedule of events for those languages which use a list of names of events as a timing control.

Simulation languages have commands to place (or delete) names of records on lists other than the schedule of events list. Because test conditions for the execution of a particular subprogram involve membership in lists, simulation languages usually have logical commands to determine whether a particular record belongs to a list, to select the first, last, or other record in a list and to perform a sequence of instructions for all members of a list if particular conditions are satisfied. Normally, the members of a list are ordered according to a specified criterion when a new number is added. (CLP permits more complicated "list" structures.) The commands used for these operations are given in lines 4, 5 and 6.

(*d*) *Programming Features.* Other features of the languages are outlined in Table III.4. GPSS differs in a major way from the other languages in its basic conceptual unit for programming. In GPSS, the structure and action of a system is described using block diagrams in which each block represents a step in the action of the system. Thirty-six specific block types are included in the language, and the system must be described by combinations of these. The other languages employ the more common and more general construction used in general purpose languages in

which the basic unit is the statement; statements may be combined to form subprograms (line 1).

All of the six languages except GPSS require some knowledge of a particular general purpose language (line 2). A SIMSCRIPT or GASP (SOL) user should know FORTRAN (ALGOL) and he can incorporate FORTRAN (ALGOL) statements in his source program. A CLP user must know CORC since it is an extension of CORC. A CSL user will find a knowledge of FORTRAN very useful. A GPSS user can make use of FAP/MAP if he needs "help."

Some language designers believe that problem formulation is easier if the language were designed to use standard flowchart symbols and decision tables. GASP and GPSS provide a flowchart convention; none of the languages considered here incorporate decision table conventions (line 3).

Because of the complexity of simulation programs, the capability of recursion can be very useful. SOL possesses the same recursion power that ALGOL does. CSL can use recursion on logical test chains. The others do not have this capability (line 4).

The arithmetic commands of the simulation language are those of the general purpose language in which it is embedded (CORC for CLP; FORTRAN for SIMSCRIPT, CSL, and GASP; ALGOL for SOL). SIMSCRIPT provides several additional commands. In GPSS several of the blocks are used to do arithmetic operations (line 5).

A simulation program must compile and summarize various statistics during a "run" or must store records of the history of a run so that the results can be analyzed in a separate operation. All six languages provide commands to collect statistics during the simulation run (line 6).

Most simulation programs involve the generation of random numbers and random variates having specified probability distributions. The capability of the various languages to do this is shown in line 7. The table look-up feature of SIMSCRIPT and GPSS for nonrectangular distributions and the calling convention of SOL are particularly powerful.

The input and output facilities are shown in line 8. In line 9 are shown the facilities to generate output reports. GASP, GPSS and SOL provide standard summaries which do not have to be programmed. SIMSCRIPT and CLP have flexible and powerful report generators but not standard summaries.

A final feature of a language is the extent to which it can be used for problems other than simulation (line 10). SIMSCRIPT is designed as a general programming package, especially suited for but not limited to simulation problems. GPSS is not usually appropriate for nonsimulation programs. The CORC general-purpose compiler is a subset of CLP and can be used for problems other than simulation. CLP is itself a list processing language and can be so used. GASP subroutines can be used in FORTRAN programs in the same way as any other FORTRAN subroutines.

(*e*) *Mechanics of Use.* In the previous four subsections we have dealt with characteristics of six simulation languages. In this subsection, the characteristics describing the packages are covered. These characteristics are more the result of decisions made by the implementer of the language than those made by the language designer and hence they may vary from one implementation to another.

A particular implementation of each language has been selected for detailed comparison. The implementer, the computers for which the package is available, the documentation, and training aids are shown in Table III.5, line 1. If there are other implementations, they are listed in line 7. It should be noted that, even if a package is available for a particular machine, the incorporation of it at a particular installation may be difficult and frustrating, especially if the implementation was designed for a different operating system.

The compilation and running procedure is described in line 2. The implementer may choose to generate an object program from the source program or to have the source program interpreted at object time. The latter is the method used by GPSS and SOL. GPSS is a FAP (MAP) assembled program. The GPSS model is executed interpretively in the same sense that control passes to the appropriate block subprograms in a sequence determined by the block diagram. In the Case Institute of Technology implementation of SOL, the first pass through the SOL compiler produces pseudocode which then is executed by an interpreter. One disadvantage of all interpretative systems compared with compilers is that more computer time for execution is usually used.

The implementer has a choice of translating the source language to a general purpose language for which a compiler already exists or to build a compiler that generates object code directly. The early implementations of SIMSCRIPT and CSL required translation to FORTRAN II; recent implementations go directly to object code. GASP is written in FORTRAN and is compiled as any other FORTRAN program. CLP is compiled directly by a "load and go" compiler. One advantage of a method based on compiling is that the object program that results from translation and compilation of a source program may be re-used to avoid retranslation and compilation on subsequent runs.

Debugging and diagnostic aids (line 3) are largely those available with the associated compiler languages. GPSS has a variety of self-contained diagnostic aids. The CORC diagnostics associated with CLP are unusually extensive and include correction of spelling and punctuation errors by the compiler and error detection during run time. This is possible because the compiler program remains in memory during program execution. SIMSCRIPT is found by many to be difficult to debug and suffers from limited diagnostic capabilities in the package itself. A trace capability, available relatively easily using the report generator, can be used to overcome some of SIMSCRIPT debugging limitations.

In simulation programs, as in other computer programs,

it is desirable to have syntax error detection and correction if possible, during compilation and error detection during run time. Interpretative systems have no particular difficulty in providing these features. CLP demonstrates that it is possible to have comprehensive error detection and correction during both compile and run time.

One characteristic that is usually important is the size of the problem that can be programmed and run. To a large extent problem size is determined when memory space is allocated and by whether data can be packed (line 4). Memory is dimensioned at "load time" in SIMSCRIPT and GPSS, in the source program for the others. Also, SIMSCRIPT permits word packing of up to 4 attributes per word for more efficient use of core memory. Packing in GPSS is not under programmer control unless FAP (MAP) "Help" routines are used. The other packages do not provide packing; in SOL, packing would depend on the ALGOL implementation in which it is imbedded. Memory allocation is not dynamic except for temporary record allocation and reallocation in SIMSCRIPT, CLP and GPSS. External tape memory can be used in all with, however, significant increase in running time.

One of the features of concern to users is the computer time required to debug, compile and execute programs. Timing comparisons have been part of computer evaluation ever since customers had to choose between two or more machines and two or more language packages. Many timing comparisons of general purpose languages against each other and against simulation languages have been reported. We believe that these results are almost always not meaningful and thus are not useful for language selection. In general, these comparisons do not provide a complete description of the problem and of the programs written and run to make the comparison. Rarely is a detailed description of the background and experience of the programmers given. Too often, the experimental design is not described at all; rarely is the statistical analysis of the results described nor is there detailed presentation of the raw data from which conclusions were drawn.

It may be that detailed timing comparisons are not justified anyway because shortest possible execution time is only one criterion. Fast execution at the expense of programming time, inconvenience and program complexity may be a poor bargain.

Obviously, a user is interested ultimately in knowing how long it will take to compile and to execute a simulation program in one language compared with others. At present, unfortunately, it is almost impossible to present a valid, quantitative evaluation of such comparisons. For the six subject languages, general purpose languages (and assembly languages) usually appear to have less compilation and execution time. Considering the state of the art, these results are not surprising. Nevertheless, there should be no essential reason why a language specifically designed for simulation *must* be less "efficient" than a general purpose language used for simulation.

Of the two most frequently used simulation languages, GPSS and SIMSCRIPT, SIMSCRIPT is generally considered to be faster in execution than GPSS II (if the SIMSCRIPT program is written properly). GPSS III is faster than GPSS II, and the differential has been reduced.

A simulation package can be particularly useful if the implementation provides features to make experimentation and model manipulation easy. In line 6, the availability of such features as initialization, parameter change and program rerunning is detailed for the six packages here discussed. Obviously, the ease or difficulty of simulation experimentation will also depend on the operating system and the operating procedure of a particular installation.

Implementations on other computers of each of the simulation languages are given in line 10.

IV. Implications for Users, Language Implementers, and Language Designers

It is apparent that none of the languages and packages discussed here is the "best" for all purposes. However, from this comparison, we can identify some implications that are relevant for users, language implementers and language designers.

1. USERS

The user would like a language that is best for formulating his model. He would like a package that is best for transforming the model into a computer program and for running the model.

An analyst may formulate his problem as a continuous-change model, a discrete change model, or a combination of the two. The selection of a language for simulating continuous-change models can be aided by study of the comparisons given by Brennan and Linebarger (1964) (1965) and by Clancy and Fineberg (1965). If the model is a discrete-change model or a combination of the two types, the analyst can choose a general purpose language or one of the special purpose simulation languages. For model formulation, the simulation languages have great advantage because they help the definer of a simulation problem crystallize his thinking and reduce model formulation time and the programs are both documentation and a means for communication.

The choice of a discrete simulation language and package will depend to a large extent on the criteria that are most important to a user:

1. If a user wants the most powerful simulation package now generally available, he should use SIMSCRIPT. If he chooses SIMSCRIPT, however, a user would be well advised to have expert help available because the language is complex and the diagnostics are limited; many have found the SIMSCRIPT manual wanting. Furthermore, a potential SIMSCRIPT user *must* know FORTRAN.

2. If the most important criterion is ease of learning and use, GPSS should be chosen. GPSS is specifically designed for new users, and no knowledge of computer operation is assumed. Its use of flowcharts is considered

by many as being particularly attractive for non-programmers; a person familiar with flowcharts usually finds GPSS not difficult.

3. If a user wishes to develop simulation capability by adding to a presently available general purpose language, he can add GASP to FORTRAN, CLP to CORC, and SOL to ALGOL. This approach will probably get him "on the air" faster than incorporating SIMSCRIPT or CSL or whatever into the operating system of his computer installation. However, the user usually pays for this by accepting restrictions in programming features and in size of problems that can be handled.

Usually, the user will be forced to use a simulation package made available by his computing facility. The management of the facility usually is responsible for incorporating the package into the operating system, for providing for the maintenance of the package, for furnishing instruction in the language, and for assisting in the use of the language. The computer center management naturally tends to choose a package that is available for the installed computer and its operating system. If there is a choice, it will choose one that is consistent with its operating philosophy and for which the implementation is easy. A computer center is not likely to place as high a value as an individual user on the value of the language as a communication tool or on the minimization of problem definition and programming time. The individual user is thereby constrained in his choice of language. If he chooses one that is not incorporated in the monitor system, he pays in longer turnaround time, little or no programming assistance and usually higher costs for computer time because of setup and teardown charges.

It may seem strange that we have not mentioned the suitability of a particular language for a user's problem as a criterion. There is as yet no conclusive evidence that one simulation language is best for a variety of problems. In fact the evidence so far seems to indicate the contrary: a number of problems have been programmed with approximately equal ease in several languages.[1]

2. LANGUAGE IMPLEMENTERS

The factors which are primarily under the control of the language implementer are those listed in Table III.5. Many of the most serious deficiencies of present day simulation packages, from the viewpoint of the user, are caused by implementation rather than design. It is therefore of considerable importance to improve language implementation. The language implementer is often a professional programmer who takes pride in writing packages that use the least amount of memory space and run in the least possible time. In doing so, he often introduces restrictions and conventions that are bothersome to the ultimate user.

We believe the greatest deficiency of presently available packages is the lack of adequate documentation and instructional material. Every language should have a basic primer, a reference manual, worked-out examples and exercises for the novice, a *complete* description of how to get a program run, elementary hints for the novice, and sophisticated hints for the experienced programmer. It may be that we should not expect such material from language developers; users and groups of users may have to do the job themselves. Whoever provides it, such material is essential and needed.

Good diagnostic aids are particularly important in simulation work because of the complexity of the models being programmed and because of their stochastic nature. Implementers must provide debugging aid during both compilation and run time. However, because run time efficiency is essential in almost all "production" simulation work, we suggest that options be provided so that debugging is available when needed but can be bypassed when desired.

The size of problems that can be accommodated by a particular implementation may depend to a great extent on the way in which the use of external memories are incorporated in the package. Implementers would be well advised to consider a feature such as SIMSCRIPT's "Record Memory" and "Restore Status" or PL/I's storage allocation statements, "ALLOCATE," "FREE," "FETCH," and "DELETE." They might also consider providing an easily used capability for interruping a run so that it can be continued at some later time.

One method used to increase the size of problem that can be accommodated in fixed-word length machines is to pack data in individual "words." Word packing makes best use of memory but usually at a cost in running time. The implementation should be designed so that any running time penalty can be avoided whenever packing is not actually used in a program. Dimension-free array specification (such as in SIMSCRIPT) and dynamic memory allocation should be provided, both for efficiency and for coding ease, again under user option. Provision should be made for breaking up larger problems into smaller ones.

A number of simulation languages have been implemented for more than one machine. There is every indication that this trend will continue and the languages like SIMSCRIPT and GPSS will be available for most of the larger commercially available computers. Such efforts should be encouraged.

3. LANGUAGE DESIGNERS

Before a designer sets out to develop a new simulation language, he should seriously consider whether a new language is really necessary. A new language, in itself, is not sufficient justification for existence; some demonstration of the usefulness of new features is necessary. Often,

[1] Special purpose simulation languages for specific applications are in existence and have considerable support and justification. See for example, MILITRAN, Systems Research Group, Inc. (1964), and UNISIM, Weber (1964). For an interesting application using SIMSCRIPT for special job shop simulator problem, see Ginsberg, Markowitz and Oldfather, "Programming by questionnaire" (1965).

user complaints about existing languages are not with the language per se but with certain features of the implementation: lack of documentation, lack of training aids, difficulties in incorporating the package into a computer center's monitor system, lack of adequate debugging facilities, and so on. Such a realization, if valid, argues very strongly for the design of a simulation based on a widely available general purpose language. All the capability of the general purpose language would be available; mechanics of use would be straightforward. The new general purpose language, PL/I, may be particularly attractive for such applications because of its input-output features, its asynchronous operations, its flexible data structures, character and part-word data manipulating capability, and its list processing and memory extending commands.

The desirable features of a simulation language that have been noted in this comparison are summarized here in the order they appear in Tables III.1–4:

(a) A language should provide for at least the five types of variables: records, fields, groups of records, arrays or system variables, and lists of record identifications. More flexibility in creating and manipulating data structures is desirable because simulation models are becoming larger and more complex. Multiple precision, Boolean, and complex variables should be considered.

(b) Ability to create and destroy temporary entities is useful and probably necessary.

(c) One improvement in simulation languages might be in a rethinking about the fundamental nature of discrete-event simulation. Whether the language is oriented toward "events" or "activities" (or "Processes" as in SOL and SIMULA) will depend on the applications the designer has in mind. It may be possible to allow for all three and let the user choose the appropriate orientation when he writes his program. Another possible improvement is in the capability for simulating parallel activities. If so, there must be a method for dealing with simultaneous events; one possibility is to allow subprograms to execute either synchronously or asynchronously, as appropriate. In simulating parallel processes, it is desirable to provide for the ability to interrupt a process and to reinstate it at a future time. Perhaps a language can be developed which could combine the features of continuous and discrete change languages.

(d) A timing or master routine should be provided, but the user should have the ability to program his own by "bypassing" or ignoring the package-supplied routine.

(e) Extensive list processing capability, as much as that in SIMSCRIPT and more, is necessary in future simulation languages. The language should permit complex tree structures as well as simple lists.

(f) The language should have aids such as flowcharts and decision tables to help in formulating complicated simulation models.

(g) Recursion adds to language power and should be available at user option. The ability to define procedures and subprograms should be extended. It is particularly necessary to maintain compatibility between a user-developed special programming language and the master general purpose language.

(h) Commands to compile statistics easily must be provided.

(i) Facility for generating variates having specified and arbitrary probability distributions is needed.

(j) User experience indicates that the user should have the option of a SIMSCRIPT-type flexible report generator and a comprehensive standard summary. The standard summary is necessary to save programming time. The report generator is needed to produce reports that can be used by management without transcription.

(k) Every possible aid to facilitate experimentation with a simulation model should be provided: for multiple runs, for changing parameter values, for changing data, for analyzing results, for stopping and restarting runs and for optimization.

RECEIVED January, 1966; REVISED June, 1966

V. References and Bibliography

1. *References*

ALLEN, R. G. D. (1965) *Mathematical Analysis for Economists.* MacMillan & Co., New York.

BEACH, E. F. (1957) *Economic Models, an Exposition.* John Wiley & Sons, New York.

BRENNAN, R. D., AND LINEBARGER, R. (1964) A survey of digital simulation. *Simulation 3*, No. 6.

BRENNAN, R. D., AND LINEBARGER, R. (1965) An evaluation of digital analogue simulator languages. Proc. IFIP Congress. Vol. 2.

CHEN, G. K. C. (1964) Private communication.

CLANCY, J. J., AND FINEBERG, MARK S. (1965) Digital simulation languages: a critique and a guide. Proc. AFIPS Fall Joint Comput. Conf., Vol. 27, pp. 23–36.

FEDDERSEN, A. P., AND O'GRADY, W. D. (1965) "Simulation: A decision device for resource allocation (Abs.). *Bull. Oper. Res. Soc. Am., 13* Suppl. 2, p. B168.

FORRESTER, J. W. (1961) *Industrial Dynamics.* MIT Press, Cambridge, Mass., and John Wiley & Sons, New York.

FREEMAN, D. E. (1964) Programming languages ease digital simulation. *Contr. Eng.* pp. 103–6a

GINSBERG, A. S. (1965) Simulation programming and analysis of results. P-3141, RAND Corp., Santa Monica.

GINSBERG, A. S., MARKOWITZ, H. M., AND OLDFATHER, P. M. (1965) Programming by questionnaire. RM-4460-PR, RAND Corp., Santa Monica.

IBM Corporation, Job shop simulator.

IBM Corporation (1966) Bibliography on simulation. Rep. 320-0924-0.

KRASNOW, H. S., AND MERIKALLIO, R. (1963) The past, present and future of general simulation languages. *Man. Sci. 11*, 236–67.

LASKI, J. G. (1965) On the time structure in Monte Carlo simulations. *Oper. Res. Quart. 16:* 3, 329–340.

MORGENTHALER, G. W. (1961) The theory and application of simulation in operations research. In *Progress in Operations Research*, Russel L. Ackoff (ED.) John Wiley & Sons, New York, pp. 363–419.

MURPHY, J. G. (1964) A comparison of the use of the GPSS and SIMSCRIPT simulation languages in designing communications network, Tech. Mem. TN—03969, Mitre Corp., Bedford, Mass.

SCRAMSTAD, H. K. (1962) Combined analog-digital techniques in simulation. *Advances in Computers, Vol. 3*. Academic Press, New York, pp. 275–298.

TOCHER, K. D. (1965) Review of simulation languages. *Oper. Res. Quart. 15*, 2, 189–218.

TUSTIN, A. (1953) *The Mechanism of Economic Systems*. Harvard U. Press, Cambridge, Mass.

YOUNG, KAREN (1963) A user's experience with three simulation languages (GPSS, SIMSCRIPT, and SIMPAC). TM-1755/000/00, Systems Development Corp., Santa Monica.

2. *Selected Bibliography on Simulation*

BLAKE, K., AND GORDON, G. (1964) Systems simulation with digital computers," *IBM Sys. J. 3*, 3, 14–20.

BRENNEN, MICHAEL E. (1963) Selective sampling—a technique for reducing sample size in simulation of decision-making problems. *J. Ind. Engng. 14*, 291–96.

BURDICK, D. S., AND NAYLOR, T. (1966) Design of computer simulation experiments for industrial systems. *Comm. ACM 9*, 5, 329–338.

CLARK, C. E. (1961) Importance sampling in Monte Carlo analyses. *Oper. Res., 9*, 603–20.

CONWAY, R. W. (1963) Some tactical problems in digital simulation," *Man. Sci. 10*, 47–61.

DEAR, R. E. (1961) Multivariate analysis of variance and covariance for simulation studies involving formal time series. Doc. FN-5644, System Development Corp., Santa Monica, 60 pp.

EHRENFELD, S., AND BEN-TUBIA, S. (1962) The efficiency of statistical simulation procedures. *Technometrics 4*, 257–275.

FISHMAN, G. S., AND KIVIAT, P. H. (1965) Spectral analysis of time series generated by simulation models. RM-4393-PR, RAND Corp., Santa Monica, 73. pp.

FISHMAN, G. S. (1966) Problems in the statistical analysis of simulation experiments: The comparison of means and the length of sample records. RM-4880-PR, RAND Corp., Santa Monica, 22 pp.

GALLIHER, HERBERT P. (1959) Simulation of random processes. *Notes on Operations Research*. Oper. Res. Center, MIT, Cambridge, Mass., pp. 231–250.

GEISLER, MURRAY A. (1962) The sizes of simulation samples required to compute certain inventory characteristics with stated precision and confidence. RM-3242-PR ASTIA no. AD, 286 796, RAND Corp., Santa Monica; also *Man. Sci. 10*, (1964), 261–86.

HAMMER, C. (1961) Computers and simulation. *Cybernetica 4*, 4; *1* (1962), 50.

HERMAN, H. H. (1961) Simulation: a survey. 1961 Western Joint Comput. Conf., 1.1, pp. 1–9.

HAWTHORNE, G. B., JR. (1964) Digital simulation and modeling. *Datamation, 10*, 25–29.

HOGGATT, A. C., AND BALDERSTON, F. E. (1964). *Symposium on Simulation Models: Methodology and Applications to the Behavioral Sciences*. South-Western Publishing Co., Cinn. Ohio.

JACOBY, J. E., AND HARRISON, S. (1962) Multi-variable experimentation and simulation models. *Naval Res. Logistics Quart. 9*, 121–136.

Lawrence Radiation Lab. (1964) Monte Carlo methods, a bibliography covering the period 1949 to 1963. UCRL-T823D, U. of California, 116 pp.

MCARTHUR, D. S. (1961) Strategy in research—alternative methods for design of experiments. *IRE Trans. Eng. Man. EN-8, 1*, 34–40.

SCHENK, H., JR. (1963) Computing 'AD ABSURDUM.' *The Nation*, June 15.

SHUBIK, MARTIN (1960) Bibliography on simulation, gaming, artificial intelligence and allied topics. *J. Am. Stat. Assoc. 55*, 736–751.

SMITH, E. C., JR. (1962) Simulation in systems engineering. *IBM Sys. J. 1*, 33–50.

THOMAS, CLAYTON J. (1961) Military gaming. *Progress in Operations Research, Vol. 1*. John Wiley & Sons, Inc., New York, pp. 421–464.

TOCHER, K. D. (1963). *The Art of Simulation*. D. Van Nostrand Co., Inc., Princeton, N. J.

U. of Michigan, Proceedings (1960) Symposium on digital simulation techniques for predicting the performance of large scale systems, May 23–25, Ann Arbor, Mich., Rep. No. 2354-33X.

YAGIL, S. (1963) Generation of input data for simulations. *IBM Sys. J. 2*, 288–296.

3. *Simulation Languages: Manuals and Specifications*

CLP:

CONWAY, R. W., MAXWELL, W. L., AND WALKER, R. J. (1963) An Instruction Manual for CORC—The Cornell Computing Language. Cornell U., Ithaca, N. Y.

MAXWELL, W. L., AND CONWAY, R. W. (1963) CPL Preliminary Manual. Dept. of Indus. Eng., Cornell U., Ithaca, N. Y., No. 3, 9580, October, 22 pp.

WALKER, W. E., AND DELFAUSSE, J. J, (1964) The Cornell list processor. Cornell U., Ithaca, N. Y.

CSL:

BUXTON, J. N., AND LASKI, J. G. (1962) Control and simulation language. Esso Petroleum Co., Ltd., and IBM United Kingdom, Ltd., London, England, August. Reprinted in the *Comput. J. 5*, 3 (1964).

IBM United Kingdom, Ltd. and Esso Petroleum Co., Ltd. (1963) Control and simulation language—introductory manual. March, 39 pp.

IBM United Kingdom, Ltd. and Esso Petroleum Co., Ltd. (1963) Control and simulation language—reference manual. March, 95 pp.

DYNAMO:

PUGH, A. L., III (1961) *DYNAMO User's Manual*. MIT Press, Cambridge, Mass.

ESP:

WILLIAM, J. W. J. (1964) E.S.P.—the Elliot simulator package. *Comput. J. 6*, 328–331.

FORSIM IV:

FAMOLARI, E. (1964) FORSIM IV simulation language user's guide. ESD-TDR-64-108, The MITRE Corp., May, 48 pp.

GASP:

BELKIN, J., AND RAO, M. R. (1965) GASP users' manual. United States Steel Corp., Appl. Res. Lab., Monroeville, Pa.

KIVIAT, P. J. (1963) GASP—a general activity simulation program. Proj. No. 90. 17-019 (2), United States Steel Corp., Appl. Res. Lab., Monroeville, Pa.

KIVIAT, P. J., AND COLKER, A. (1964) GASP-a general activity simulation program. P-2864 RAND Corp., Santa Monica.

GPSS:

EFRON, R., AND GORDON, G. (1964) A general purpose digital simulator and examples of its application: Part I—description of the simulator. *IBM Sys. J. 3*, 1, 22–34.

GORDON, G. (1962) A general purpose systems simulator, *IBM Sys. J. 1*, Sept., 18–32.

GORDON, G. (1961) A general purpose systems simulator program. 1961 Eastern Joint Computer Conf., pp. 87–104.

HUROWITZ, M. (1965) General purpose systems simulator II, GPSS II on the UNIVAC 1107, preliminary user manual. Univac Div. of Sperry Rand Corp.

IBM Corporation. (1963) Reference manual, General Purpose Systems Simulator II. B 20-6346 149 pp.

GSP:

TOCHER, K. D. Handbook of the general simulation program. Vol. I (revised) and Vol. II. Dept. Operat. Res. Cybernetics Rep. 77/ORC 3/ Tech. and Rep. 88/ORC 3 Tech, The United Steel Companies, Ltd., Sheffield, England.

TOCHER, K. D., AND OWEN, D. G. (1960) The automatic programming of simulators. *Proc. Second Intern. Conf. Operat. Res.* English Universities Press, 1960, p. 50.

MILITRAN:

Systems Research Group, Inc. (1964) MILITRAN reference manual.

Systems Research Group, Inc. (1964) MILITRAN programming manual.

Systems Research Group, Inc. (1964) MILITRAN operation manual.

OPS:

GREENBERGER, M. (1964) The OPS-1 manual. Project MAC, MIT, Cambridge, Mass., MAC—TR-8.

GREENBERGER, M. (1964) A new methodology for computer simulation. Project MAC, MIT, Cambridge, Mass., MAC-TR-13.

GREENBERGER, M., JONES, M. M., MORRIS, J. H., JR., AND NESS, D. N. (1966) *On-Line Computation and Simulation: The OPS-3 System.* MIT Press, Cambridge, Mass.

QUIKSCRIPT:

TONGE, F. M., KEELER, P., AND NEWELL, A. (1965) QUIKSCRIPT, a SIMSCRIPT-like language for the G-20. *Comm.* 1959 *ACM 8*, 6, 350–354.

SIMCOM:

SANBORN, T. G. (1959) "SIMCOM—the simulation compiler. Eastern Joint Comput. Conf., pp. 139.

SIMPAC:

BENNETT, R. P., COOKEY, P. R., HOVEY, S. W., KRIBS, C. A., AND LACKNER, M. R. (1962) SIMPAC user's manual. TM-602/000/00, Sys. Develop. Corp., Santa Monica, April 15.

LACKNER, M. R. (1962) Toward a general simulation capability. 1962 Western Joint Comput. Conf. p. 1-14.

SIMSCRIPT:

DIMSDALE, B., AND MARKOWITZ, H. M. (1964) A description of the SIMSCRIPT language. *IBM Sys. J. 3*, 1, 57–67.

GEISLER, M. A., AND MARKOWITZ, H. M. (1963) A brief review of SIMSCRIPT as a simulating technique. RM-3778-PR, RAND Corp., Santa Monica.

HAUSER, B., AND MARKOWITZ, H. M. (1963) Technical appendix on the SIMSCRIPT simulation programming language. RM-3813-PR, RAND Corp., Santa Monica.

MARKOWITZ, H., HAUSNER, B., AND KARR, H. (1962) SIMSCRIPT: A simulation programming language. RAND Mem. RM-3310-PR, RAND Corp., Santa Monica; also published, Prentice Hall, Englewood Cliffs, N. J.

SIMULA:

AMIRY, A. P., AND TOCHER, K. D. (1963) New developments in simulation. Proc. Third Internat. Conf. Operat. Res., pp. 832, 658.6, 161.

DAHL, O. J., AND NYGAARD, K. (1963) Preliminary presentation of the SIMULA language (as of May 18th, 1963), and some examples of network descriptions. Norwegian Comput. Center, Forsknongsveien, 1B, Oslo, Norway.

DAHL, O. J., AND NYGAARD, K. (1965) SIMULA: A language for programming and description of discrete event system, introduction and user's manual. Norwegian Comput. Center, Forsknongsveien 1B, Oslo, Norway.

NYGAARD, KRISTEN. (1963) A status report on SIMULA.—a language for the description of discrete event network. Proc. Third Intern. Conf. Operat. Res., p. 825, 658.6, 161.

NYGAARD, K. (1962) SIMULA—An extension of ALGOL to the description of discrete event networks. Paper given at the IFIPS Conference, Munich, Aug. 1962.

NYGAARD, K., (1963) SIMULA—an extension of ALGOL to the description of discrete-event networks. Information Processing, 1962; Proc. IFIP Congress 62, North-Holland Publishing Co., Amsterdam, pp. 520–522.

SOL:

KNUTH, DONALD E., AND MCNELEY, J. L. (1963) SOL—A symbolic language for general-purpose systems simulation. 45 pp.

KNUTH, DONALD E., AND MCNELEY, J. L. (1964) SOL—A symbolic language for general purpose systems simulation. *Trans. IEEE* pp. 401–414.

KNUTH, DONALD E., AND MCNELEY, J. L. (1964) A formal definition of SOL. *Proc. IEEE.*

UNISM:

WEBER, J. H., AND GIMPELSON, L. A. (1964) UNISM—A simulation program for communications networks. Proc. AFIPS 1964 Fall Joint Comput. Conf., pp. 233–249.