INTRODUCTION TO SIMULATION LANGUAGES*

Robert E. Shannon
University of Alabama in Huntsville
Huntsville, Alabama

INTRODUCTION

Early effort in a simulation study is concerned with defining the system to be modeled and describing it in terms of logic flow diagrams and functional relationships. But eventually one is faced with the problem of describing the model in a language acceptable to the computer to be used. Most digital computers operate in a binary method of data representation, or in some multiple of binary such as octal or hexadecimal. Since these are awkward languages for users to communicate with, programming languages have evolved to make it easier to converse with the computer. Unfortunately, so many general and special purpose programming languages have been developed over the years, that it is a nearly impossible task to decide which language best fits or is even a near best fit to any particular application. Over 170 programming languages were in use in the United States in 1972 [1] and today there are even more. Consequently, the usual procedure is to use a language known by the analyst, not because it is best, but because it is known. It should be stated that any general algorithmic language is capable of expressing the desired model; however, one of the specialized simulation languages may have very distinct advantages in terms of ease, efficiency and effectiveness of use.

It is not the purpose of this paper to teach how to program in any of the languages described, nor to discuss implementation techniques. What we do hope to accomplish is to make the reader aware of the characteristics of some of the more popular languages, their strengths and weaknesses. The major differences between special purpose simulation languages in general are: (1) the organization of time and activities, (2) the naming and structuring of entities within the model, (3) the testing of activities and conditions between elements, (4) the types of statistical tests possible on the data and (5) the ease of changing model structure. In the following sections we intend to provide comparisons of several languages after first showing the various philosophies of language design and describing a number of key factors involved in choosing a language.

ADVANTAGES OF SIMULATION LANGUAGES

The development of simulation languages has been an evolutionary process which began in the late 1950's. At first the languages used in simulation were general purpose languages. After programming a number of models, analysts recognized that many of the situations being simulated could be categorized broadly as systems involving the flow of items through processes. Since many of the programs had functionally similar processes, the idea developed almost simultaneously within several groups of researchers in the late 1950's and the early 1960's to develop special purpose languages. These developed gradually from assembly language programs with special features, through extended commercially available problem oriented languages, to sophisticated special purpose simulation languages. Any algorithmic programming language can be used for simulation modeling. But those languages designed specifically for the purpose of computer simulation provide certain useful features. These include:

(1) Reduction of the programming task

(2) Provision of conceptual guidance

(3) Aide in defining the classes of entities within the system

(4) Flexibility for change

(5) Provide a means of diferentiating between entities of the same class by characteristic attributes or properties

(6) Relate the entities to one another and to their common environment

(7) Adjust the number of entities as conditions vary within the system.

Emshoff and Sisson [2] believe that all simulations require certain common functions, which make a simulation language different from a general

* This paper is a distillation of material appearing in Shannon, Robert E., System Simulation: The Art and Science, Prentice-Hall, Inc., Englewood, New Jersey, 1975.

algebraic or business programming language. Among these are the need to:

(1) Create random numbers

(2) Create random variates

(3) Advance time, either by one unit or to the next event

(4) Record data for output

(5) Perform statistical analyses on recorded data

(6) Arrange outputs in specified formats

(7) Detect and report logical inconsistencies and other error conditions.

Further, they state that for simulations in which discrete items are processed by specific operations, the following common processes are additionally present:

(1) Determine type of event (after retrieval from an event list)

(2) Call subroutines to adjust the state variables as a result of the event

(3) Identify specific state conditions

(4) Store and retrieve data from lists (tables or arrays), including the event list and those that represent the state.

Some of the simulation languages are languages in the more general sense that, beyond linking the user with the computer as a means of conversing, they afford the user an aid to problem formulation. Having a vocabulary and a syntax, they are descriptive, and consequently their users tend after some utilization (as with other languages) to think in them. Thus Kiviat [3] believes the two most important reasons for utilizing simulation languages as opposed to general purpose languages, are programming convenience and concept articulation. Concept articulation is important in the modeling phase and in the overall approach taken to system experimentation. Program convenience points up its importance during the actual writing of the computer program. Another advantage of the simulation languages is their use as communication and documentation devices. By writing in English-like languages, simulations can more easily be explained to project managers and other non-programming-oriented users. A major cited disadvantage of using simulation languages is that most were developed by individual organizations for their own purpose and released to the public more as a convenience and intellectual gesture than a marketed commodity. Since this was and still is to a large degree the case, most users, accustomed to having computer manufacturers do the compiler support work as a service, are not set up to do this work themselves. More and more, however, well-documented simulation languages are commercially available.

FACTORS PERTINENT TO LANGUAGE SELECTION

Before a programming language is selected, the computer to be used (both as to type and model) must be determined. Ideally, selection of the computer to be used is one of the decision options open to the analyst. In actual practice, the user probable has available a particular hardware configuration and little latitude as to modification or choice. Once the computer to be used is known, we are ready to select the language. This selection should be a two-phased screening process. The initial phase can be accomplished at any time, even before a particular problem arises. In this phase, language possibilities are examined for their operational characteristics relative to the user's environment and capabilities. The second phase, which is related to the specific problem must be accomplished after subsystem modeling and computer selection.

In the first phase of the selection process we are concerned with the availability of references, documentation and software compatability. We are basically trying to screen the multiplicity of available languages to find those that make good sense for us to consider later when we have a specific problem. The type of questions to be answered deal with the general environment in which the analyst finds himself. Among the questions we need to explore are:

(1) Are intelligibly written user's manuals available?

(2) Is the language compiler compatible with available computer systems?

(3) Is this language available on other computer systems where the user's problem might be run?

(4) Does the language translator provide documentation and extensive error diagnostics?

(5) When the organizing, programming, and debugging time is combined with the compiling and running time does the efficiency appear attractive?

(6) What is the cost of installing, maintaining and updating the software for the language? (Since some languages are proprietary, there may be an explicit charge for these services).

(7) Is the language already known or easily learned?

(8) Are a sufficient number of simulation studies anticipated for the future to justify the cost of learning and installing the new language?

In the second phase, we must deal with the characteristics of the specific problem at hand. Several different languages have probably survived the Phase I screening and are now available for possible use. Phase II, therefore, deals with choosing the specific language to be used on the specific problem at hand, with the specific computer to be used. Among the issues to be dealt with in this phase are:

(1) What is the range and applicability of of the world view of the language?
   (a) What are its time advance methods?
   (b) Is it event, activity, or process oriented?
   (c) What is its random number and random variate generation capability?

(2) How easily can state and entity variable data be stored and retieved?

(3) What is the flexibility and power provided by the language to modify the state of the system?

(4) How easily can it be used to specify dynamic behavior?

(5) What are the forms of output available, what are their utility and what statistical analyses can be performed on the data?

(6) How easy is it to insert user-written sub-routines?

The term "world view" appears in many publications describing simulation languages. It describes the way the language designer conceptualized the systems to be modeled using that language. Each simulation language has such an implicit view of the world which must be invoked when using it. The world view of a typical discrete--change simulation language might be expressed as:

(1) The world is viewed as a set of entities which may be modified or qualified by their characteristics called attributes.

(2) The entities interact with specific activities of the world consistent with certain conditions which determine the sequence of interactions.

(3) These interactions are regarded as events in the system which result in changes in the state-of-the-system.

The concepts of process, activity and event are shown in Figure 1 which shows the sequence of events for washing a car. The process of washing a car consists of 3 activities, namely, vacuumming washing and drying. The beginning and finish of each of these activities constitutes an event. The car itself would be the entity of concern.

Since most simulation studies are concerned with a system's performance over a period of time, one of the most important considerations in designing the model and choosing the language in which to program it is the method used for timekeeping. Timekeeping in a simulation has two aspects or functions: (1) advancing time or updating the time status of the system and (2) providing synchronization of the various elements and occurrence of events. Two basic timekeeping mechanisms are available for use; the fixed time increment and the variable time increment methods [4]. They are also sometimes referred to as fixed time step and next event step, respectively. The fixed increment method updates the time in the system at predetermined, fixed length time intervals (the simulation walks through time with a fixed stride). The next event or variable time increment method, on the other hand, updates at the occurrance of each significant event, independent of the time elapsing between events (the simulation walks through time on events). Very little research has been accomplished which guides the analyst in choosing between fixed increment vs. next event advance methods. Conway, Johnson and Maxwell[5] provide some guidance, as does Lave[6], Chu and Naylor[7] and Bradley[8]. Some simulation languages restrict the user to either fixed increment or next event time flow mechanisms, whereas others allow the use of either. We can offer no hard and fast rules as to when fixed increment versus next event timekeeping is preferred. Under certain sets of circumstances each shows distinct strengths and advantages. The final decision depends upon the nature of the particular system being modeled. But in general we should consider a fixed time increment method when:

(1) Events occur in a regular and fairly equally spaced manner.

(2) A large number of events occur during some simulated time T and the mean length of events is short.

(3) The exact nature of the significant events are not well known such as in the early part of a study.

On the other hand, the next event timekeeping method:

(1) Saves computer time when the system is static (i.e. no significant events occurring) for long periods of time.

(2) Requires no decision as to the size of time increment to use (which affects both computation time and accuracy).

(3) Is advantageous when events occur unevenly in time and/or the mean length of events is long.

LANGUAGE CLASSIFICATION

Many writers find it convenient to classify simulation models into two major categories: 1) continuous change models or 2) discrete change models. Continuous change models use fixed increment time advance mechanisms and are appropriate when the analyst considers the system he is study-

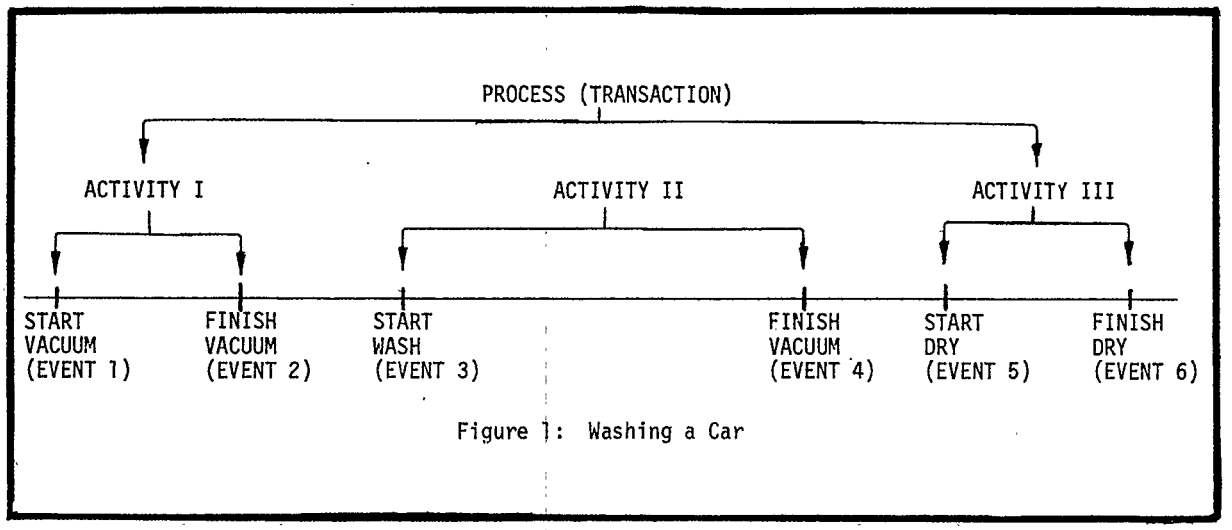

Figure 1: Washing a Car

ing as consisting of a continuous flow of information or items counted in the aggregate rather than as individual items. In discrete change models, the analyst is interested in what happens to individual items in the system. Most discrete change models, therefore, utilize the next event type of timekeeping. Some problems are clearly described best by one type or the other, whereas either type might be used for other problems.

In the most general sense, there are three computer techniques available for simulation; digital, analog and hybrid. One possible classification scheme is depicted in Figure 2. There are several versions and dialects of many of these languages and therefore only generic or family names have been used instead of listing all the various versions. Appropriate references for each of the languages shown may be found in any of the following [2(Chapter 4), 4(Chapter 3), 1, 3, 9 10,11].

The continuous-change block oriented simulation languages are all descendants of the work of Selfridge [12] in 1955. His program was unnamed; however, it has proved to be the inspiration for the large field of analog-like simulation languages based upon differential equations. The languages emulate the behavior of analog and hybrid computers on a component by component basis. The analog simulator languages all draw their inspiration and motivation from the analog block diagram as a simple and convenient means for describing continuous systems.

The equation based languages break away from the restrictions impose by the complete block construction of the analog simulator languages and deal directly with the equations. In 1966 the Simulation Software Committee of Simulation Councils, Inc., presented preliminary specifications for a Continuous System Simulation Language. The purpose was to standardize the language format and structure of digital analog simulator programs. It was hoped that future benefits would be comparable to those achieved by the by the American Standards Association, Standards Committee on FORTRAN. Just as there are considered to be first and second generation discrete-change languages, the continuous-change languages published after these specifications showed a different orientation and have a direct equation orientation.

Kiviat [3] traces the separation of early discrete-change simulation theory into two schools. These were the schools introduced by IBM with their GPSS language that used flow chart symbols as basic model descriptors and the other school which is statement oriented. Flow chart languages are easier to learn, but the statement oriented languages are more flexible. Most of the new languages are statement languages even though flow chart language is appealing and in addition to GPSS, has been used in SIMCOM and BOSS. In our classification scheme, we have used the four subcategories of activity, event, process and transaction flow orientations. Transaction-flow languages are actually process languages, since they take a synoptic view of systems; however, we have established them as a separate category because of their flow chart orientation. Event, activity and process languages(with the exception of SIMCOM) all use programming statement to describe cause and effect relationships between the system elements.

One of the most interesting recent developments is the appearance of simulation languages which allow combined discrete-continuous models. GASP IV was the first well-documented language of this type although a version of SIMSCRIPT II.5 designated C-SIMSCRIPT is now also available. These languages allow the modeler to design and execute continuous, descrete or combined simulation models.

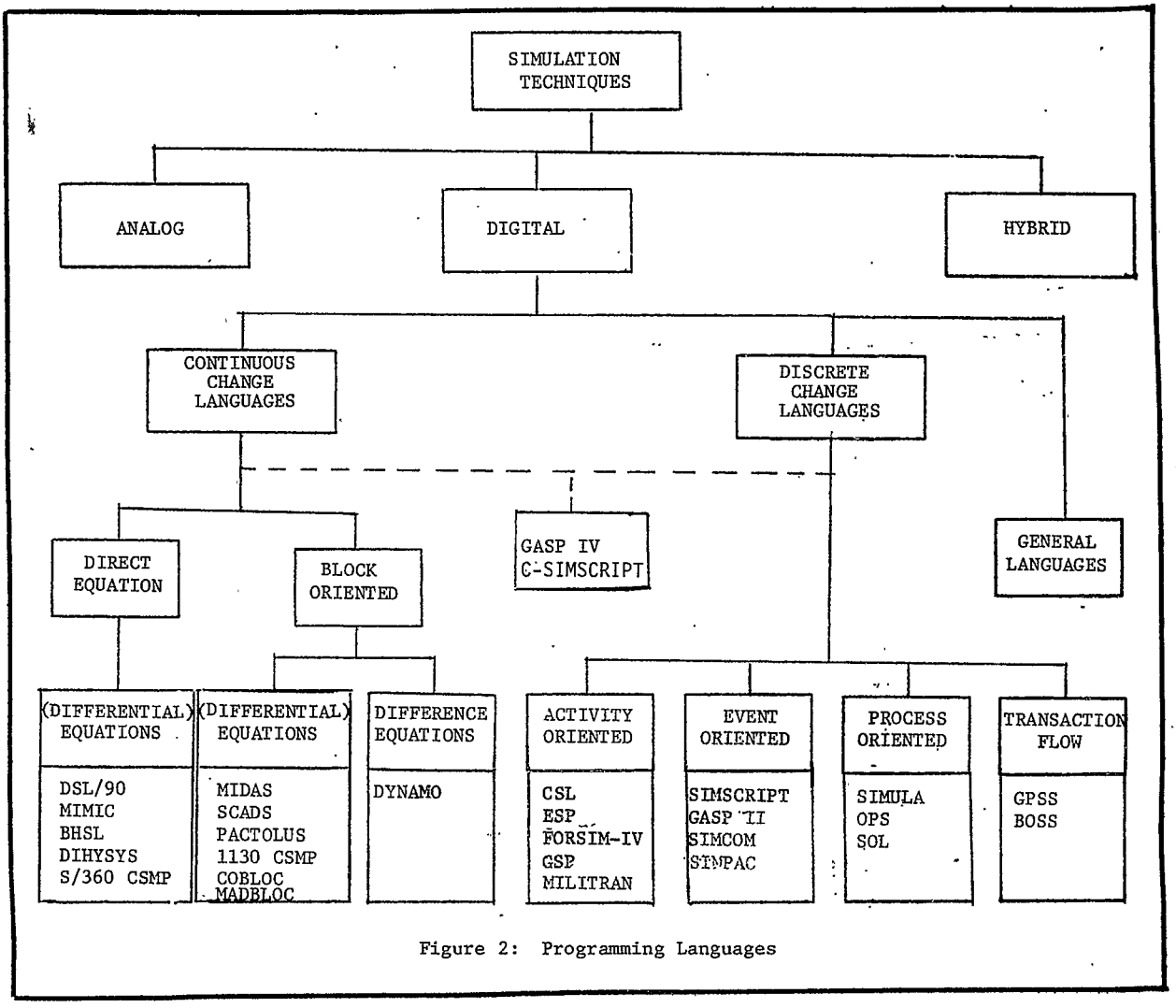

Figure 2: Programming Languages

CONCLUDING REMARKS

The three most popular discrete simulation languages in the United States appear to be GPSS, SIMSCRIPT and GASP while SIMULA appears to be the favorite in Europe. A brief summary of some of the characteristics of these languages is given below:

GPSS - General Purpose System Simulator

Initial Designer - G. Gordon I.B.M.

Versions: GPSS I, GPSS II, GPSS III, GPSS/360, GPSS-1100, GPSS V, GPSS/NORDEN plus at least 6 others

Orientation: Process (transaction)

Data Structure: Dynamic

Implementation: Assembly language

Time advance: Next event

Computers: Some version available for most computers.

SIMSCRIPT - (No known specific meaning)

Initial Designers: H. M. Markowitz, H. W. Karr and B. Hausner - RAND

Versions: SIMSCRIPT I, SIMSCRIPT I.5, SIMSCRIPT II, SIMSCRIPT II.5, C - SIMSCRIPT

Orientation: Event (secondary process)

Data Structure: Dynamic

Implementation: FORTRAN(early versions) Assembly

Time Advance: Next event for discrete fixed time step for continuous (C - SIMSCRIPT)

Computers: CDC 6000/7000
Univac 1100 series
IBM 360/370 series
Honeywell 600/6000 series
(earlier versions of SIMSCRIPT available for smaller machines)

GASP - General Activity Simulation Program

Initial Designer - P. J. Kiviat & A. Colher-RAND (A.A.B. Pritsker and Kiviat - GASP II)

Versions: GASP II, GASP IV, GASP-PLUS

Orientation: Event

Data Structure: Space Reserved

Implementation: FORTRAN, PL/I

Time Advance: Next event for discrete, fixed time step for continuous (GASP IV and PLUS)

Computers: Any computer with a FORTRAN or PL/I compiler

SIMULA - Simulation Language

Initial Designers: O. J. Dahl and K. Nygaad Norwegian Computing Center

Versions: SIMULA, SIMULA - 67

Orientation: Process

Data Structure: Dynamic

Implementation: ALGOL

Time Advance: Next event

Computer: UNIVAC 1100 series
Burroughs B5500
CDC 6000/7000 series

Each of these languages has its own strengths and weaknesses and it cannot be said that one is superior to another. As a generality, the easier a language is to learn and use--the less flexible and efficient it will be. Deciding which language is best for a specific project under a specific set of circumstances is not an easy problem owing to the large number of special and general purpose languages available. In our experience, most analysts and professional computer programmers know from one to three languages. The question then naturally arises as to whether the benefits to be derived outweigh the efforts required to learn a new language. Each person must answer that question for himself. Shannon [4] presents a decision flow diagram to be used in conjunction with a scramble book to help guide the analyst to a particular language. Other surveys or discussions that the reader might find helpful in making such a decision may be found in Emshoff and Sisson[2], Kiviat [3], Linebarger and Brennan [9], Sammett [1] Teichroew and Lubin [10] and Tocher [11].

REFERENCES

1. Sammett, J. E., "Programming Languages: History and Future," Communications of the ACM, Vol. 15, No. 7, July 1972.

2. Emshoff, J. R. and R. L. Sisson, Design and Use of Computer Simulation Models, The McMillan Co., New York, 1970.

3. Kiviat, P. J., "Development of Discrete Digital Simulation Languages," Simulation, Vol. VIII, No. 2, Feb. 1967.

4. Shannon, R. E., Systems Simulation: The Art and Science, Prentice-Hall Inc., Englewood Cliffs, N. J. 1975.

5. Conway, R. W., B. M. Johnson and W. L. Maxwell, "Some Problems of Digital Systems Simulation," Management Science, Vol. 6, October 1959.

6. Lave, R. E. Jr., "Timekeeping for Simulation," The Journal of Industrial Engineering, Vol. 17, No. 7, July 1967.

7. Chu, K. and T. H. Naylor,"Two Alternative Methods for Simulating Waiting Line Models," Journal of Industrial Engineering, Vol. 16, No. 6, November 1965.

8. Bradley, C. E., "A Variable Time-Increment Method of Queue Simulation," AIIE Transactions, Vol. 5, No. 1, March 1973.

9. Linebarger, R. N. and R. D. Brennan, "A Survey of Digital Simulation - Digital Analog Simulator Programs," Simulation, Vol. 3, No. 6, December 1964.

10. Teichroew, D. and J. F. Lubin, "Computer Simulation - Discussion of the Technique and Comparison of Languages," Communications of the ACM, Vol. 9, No. 10, October 1966.

11. Tocher, K. D., "Review of Simulation Languages," Operations Research Quarterly, Vol. 16, June 1965.

12. Brennan, R. D., "Continuous System Modeling Programs: State-of-the-Art and Prospectus for Development," Proceedings of the IFP Working Conference on Simulation Programming Languages, 1968.