COMPUTER SIMULATION PROGRAMMING LANGUAGES: PERSPECTIVE AND PROGNOSIS

Philip J. Kiviat

September 1967

P-3599

COMPUTER SIMULATION PROGRAMMING LANGUAGES: PERSPECTIVE AND PROGNOSIS

Philip J. Kiviat*

The RAND Corporation, Santa Monica, California

Simulation programming languages have been going through rapid evolutionary changes. Before 1959 there were no simulation languages--there were only simulation programs. Since 1959, when the first programming languages designed especially for simulation appeared, many different simulation modeling and programming systems have been proposed. At least five unequally different modeling schemes have found widespread acceptance and use. A large amount of modeling and programming experience has been accumulated which simulation language designers are now taking full advantage of. Papers recently presented in Oslo, Norway at the IFIP Working Conference on Simulation Languages reflect a technical sophistication that was not present a few years ago.

In this paper we first discuss some theories of simulation modeling and programming. We then describe the design aims and a few of the

---

*Any views expressed in this paper are those of the author. They should not be interpreted as reflecting the view of The RAND Corporation or the official opinion or policy of any of its governmental or private research sponsors. Papers are reproduced by The RAND Corporation as a courtesy to members of its staff.

language facilities of several "second generation" simulation programming languages. Finally, we comment on a probable future for simulation languages and simulation programming.

PERSPECTIVE

While most of the original simulation programming languages differed from one another in many small ways, there were two substantial differences between them: their modeling concepts and their implementation schemes.

Modeling Concepts

Different approaches to modeling immediately divided simulation languages into four groups: languages that focused on transaction flows, events, activities and processes.

Transaction flow languages such as GPSS have a flow chart orientation. They model systems by tracing flows of transactions through stylized activity blocks; as the transactions pass through the blocks simulated time advances, decisions are made and the logic of a simulated system is reproduced. A Programmer constructs a simulation model in a transaction flow language by assembling specialized blocks into structures that represent the logic and flow of a real or proposed system. This type of modeling is easily taught to persons who are familiar with flow chart concepts. Since the modeling blocks are also the basic programming statements, construction of a model is equivalent to writing a program.

Event, activity, and process languages on the other hand use programming statements to describe cause and effect relationships between

system elements. While flow charts are often used to describe the logic of systems modeled in these languages, they are not basic to their programming or theory. A Programmer constructs a simulation model in such a language by writing statements that define conditions that must hold for certain actions to take place, that describe the results of these actions, and that specify time relationships between system elements and activities in which they participate. In the three modeling schools within this "cause and effect" group, each advocates a different method for representing the dynamics of system behavior.

The "event" school typified by SIMSCRIPT represents each time-dependent act in a simulation model as an instantaneous occurrence in simulated time that is scheduled to occur when it is known within the dynamics of the model that the proper conditions exist. These languages have statements such as CAUSE, SCHEDULE and CANCEL that, used along with decision-making statements, handle the scheduling of events. An executive program automatically sequences the scheduled events so that they occur properly in simulated time.

The "activity" school typified by CSL also represents time-dependent acts as instantaneous occurrences in simulated time, but believes that these occurrences should not be directly scheduled to occur within a program. Instead, conditions are specified under which they can take place. There are no "activity scheduling" statements in these languages, but executive programs that scan sets of conditions before each simulation time advance to see whether or not any activities can take place.

The "process" school typified by SIMULA advocates a concept that embraces both the event and activity formulations. A process is an

entity that exists over time and can have a dynamic behavior. Processes can be scheduled to occur, be interrupted, have subprocesses that obey them, be programmed so that they delay themselves and other processes until certain conditions are met, etc. The process concept extends the tasks of an executive program even further than the event and activity organizations. Whenever a Programmer has to do less, it is because a program has to do more.

The concepts of event, activity and process are intimately related. Various languages implement aspects of each: SIMSCRIPT operates on a pure event basis, CSL takes a pure activity approach, GSP combines the two and attempts to obtain the execution advantages of one and the modeling advantages of the other; SIMULA advocates a process concept but contains statements that permit direct event scheduling; SOL puts the transaction flow concepts of GPSS in a statement language framework and adds process concepts. Each language attempts to be unique and offer power and convenience not available in other languages. Most languages are able to "simulate" modeling mechanisms different from their own, indicating that there is often more appearance than substance in their individual claims of distinctiveness.

Modeling also encompasses data description, and simulation languages differ in the facilities they offer for describing system data structures. While facilities vary, they generally serve the same ends: a way of identifying system objects (entities, elements, transactions, machines), a way of ascribing characteristics or properties to these objects (attributes, parameters, variables) and a way of organizing objects into collections (sets, queues, groups, classes).

The differences between the data descriptions permitted are due both to the preferences of the language designers and the dictates of the compiler, if any, in which the language is implemented. Thus SIMULA has certain language features because of its ALGOL block structure, and SIMSCRIPT looks as it does because it was molded in a FORTRAN, and not an ALGOL framework.

The principal ways in which simulation languages differ is in the approaches they take to timing (events, activities, processes) and to describing data structures (static versus dynamic storage allocation, levels of indirect referencing permitted, ability to form complex data structures). Each language has been created in an environment that makes its particular features attractive; each has been the product of its designer's imagination, talent and experience. Although the languages differ greatly in their ability to handle particular types of simulation problems, each is generally able to model and simulate any kind of system. Of course, many models are difficult to construct and time-consuming to operate but that is not the point.

To date, simulation languages have not been selected because of their more subtle features but their availability. All languages are not available on all computers. SIMSCRIPT was designed for the IBM 7090/94, as were CSL and GPSS; SIMULA runs on UNIVAC 1107 or 1108; SOL's translator was written for the Burroughs 5000; GSP is available on the Feranti Pegasus and Ellit 503 computers; and so on. Only recently has SIMSCRIPT been made available (as SIMSCRIPT I.5) on a variety of manufacturers computers, has GPSS been programmed for the IBM 7040/44 and 360 computer series, and has CSL been made available on the Honeywell Series 200 computer.

Implementation

Implementation is the factor that has probably had more to do with the use of simulation languages than anything else. For a language will not, or cannot, find acceptance if its implementation makes it difficult or impossible to use. An important aspect of implementation for a simulation language is the selection of a programming technique to translate a model (source language programs) into executable machine code.

The first simulation computer programs were written in basic machine codes or assembly languages and were specialized; they were hand-tailored for the problems they were designed to solve and were often quite artful and complex. When compiler languages such as FORTRAN came along, large numbers of people switched from basic machine or assembly language coding to these easier-to-write languages.

Simulation analysts soon learned, however, that it was not easy to live this way. For one thing, Programmers were constantly rediscovering techniques that were common to all simulation programs. Also, models coded to set initial specifications were hard to use. Simulation is an experimental technique requiring iterative development of models and theories, and repeated computer runs under varying test conditions. Special purpose models were not flexible enough; changes in models were slow and expensive, and it often took longer to state a problem, devise a modeling scheme, code the model and test it than the problem environment could afford.

The first step taken to alleviate the modeling problem and make the problem solver's life a little easier was the development of general

programs designed around particular problem areas. There were job shop simulators, inventory management simulators, and military operations simulators. The intent of the designers of these simulators was to incorporate a great deal of planning and design into one general model that could be adapted to a large number of situations. While these programs were undoubtedly of great value to their designers, the idea of general purpose models did not meet with great success. The basic difficulty was that such models were not easily adapted to problems outside their design range. In some ways, this was due to the complexity of the models, and in some ways due to the programming languages used.

While these general-purpose programs were being written and studied, computer programming languages were getting more sophisticated. Persons, familiar with simulation problems and with the types of problems encountered in programming simulation models, were beginning to develop special simulation programming languages. One of the first of these was the Montecode interpretive language developed in England in 1959. In 1959 and the early 1960's, a number of special simulation programming languages were developed in this country and in Great Britain.

From its inception, GPSS, one of the earliest American simulation languages, has been interpretive. A program coded in the GPSS format is first assembled into a number of tables and then the system logic, embedded in these tables, is executed interpretively by a control program. While the language allows a programmer to go outside the symbolic GPSS language, the extent to which he can do so depends on the specific implementation. Use of GPSS is limited to computers that have GPSS

assembly and interpretive programs. Interestingly, one computer manufacturer (UNIVAC) has written a GPSS interpretive system in FORTRAN.

SIMULA, on the other hand, is a direct extension of a general-purpose compiler. Being an extension of ALGOL, a SIMULA programmer uses ALGOL augmented concepts to define his models. SIMULA is potentially available to any installation that can modify its ALGOL compiler. While this is easier to say than do, it is possible and so I hear, has been done at several universities.

While there are no FORTRAN extended compilers, there are many languages that use FORTRAN. In their original versions, SIMSCRIPT and CSL were source language to FORTRAN translators. Programs written in SIMSCRIPT and CSL were first transcribed into FORTRAN and then compiled. A programmer could use the simulation language and the FORTRAN language; he paid the price of two separate translation passes and of only being able to use SIMSCRIPT and CSL on computers that had the source language to FORTRAN translators. Recent versions of these languages, SIMSCRIPT I.5, Extended CSL and CSL-2, no longer go through this two-fold process, but compile directly into assembly code. GASP and FORSIM IV took a different approach to implementation. They used FORTRAN as a programming language to prepare subroutines and functions that carry out most of the tasks provided by the statements of the simulation languages SIMSCRIPT and CSL. While a Programmer cannot write programs as neatly in these languages as he can in a true simulation language nor does he have their power, if he has a FORTRAN compiler he has a simulation capability, and that is their advantage.

Numerous other languages use variations of these programming schemes. Unfortunately most of their implementations restrict them to one or at best a few computers. Even today, most users do not have convenient access to many simulation languages. Many people laud individual languages but cannot use them; a familiar comment has been and still is, "I'd use language X but I can't get a translator for my machine."

Another important aspect of implementation concerns the structure of simulation languages. GPSS, the most graphic of the languages, does not allow flow chart symbols to be entered directly as source language inputs, but uses symbolic codings to describe its flow chart symbols. SIMCOM has the Programmer keypunch from forms that describe the flow of information and the computations carried out within a model. Unfortunately, there are no truly graphic simulation languages, but merely different types of symbolic input languages. Some are highly readable, some are hieroglyphic; some use the formats of other, more general, languages and some exist in formats that are entirely their own.

Often a language designer has little or no choice of the form his language statements take; for example, SIMULA is constrained by ALGOL and GASP and SIMSCRIPT must obey FORTRAN conventions. On the other hand, creators of new programming languages are given the opportunity to design language statements in whatever format they choose.

Recent surveys indicate that most simulation modeling and programming is done by analysts or engineers and not by professional Programmers. The reasons for this are not clear, but it is usually pointed out that it is easier to do it this way because of "communication

problems." These problems are most likely caused by difficulties in describing system models to Programmers in a quick and unambiguous way, and by the hard fact that it is exceedingly difficult to debug simulation programs that contain logical errors unless you are an expert in both the subject area and the program. Until a better method is found, e.g., conversational modeling through graphics and time-sharing, it seems that the best direction a simulation programming language can take is toward natural language description, i.e., having a language translator convert an English-like problem description into an executable computer program.

Generally speaking, programming languages are getting more readable; COBOL and PL/I are both more readable than FORTRAN. As they become higher-level and easier to use, languages become useful for conceptualization and documentation as well as programming. The trend in all language areas: information retrieval, automatic test equipment instruction, and data processing to name a few, is away from terseness and toward readability and descriptive power.

But languages must not only be easy to write, they must embody an "adequate" theory of simulation programming. Since we are concerned with both modeling and experimentation, "adequate" must refer to both problem formulation and program execution. Laski recognized that the activity and event orientations of the British and American simulation languages each had their advantages and tried to develop a mechanism to obtain the advantages of both, but his result was a scheme and not a theory. Lackner developed a Calculus of Change that describes the dynamics of systems change, but his language is complex and probably

not suitable for general use. The process concept is being implemented in many languages today and may be a starting point for a universal theory of simulation modeling. Such a theory will have to have a capability for defining and portraying the dynamics of system change, and an efficient programming implementation.

PROGNOSIS

Simulation, as all computer-oriented technologies, is affected by advances in hardware and software. In addition to developing more broadly based modeling concepts, simulation language researchers are making full use of new programming languages and hardware-software systems.

Modeling Concepts

Modeling, as we have defined it, is concerned with structural and time-dependent descriptions of systems. Modeling research has as its foremost aim the removal of barriers to the natural description of system operations.

We have already mentioned the three basic approaches taken to modeling time-dependence in simulation models: the event, the activity and the process. It does not look as though any new approaches are on the horizon. Rather than search for new overall structures, designers are focusing attention on two major timing problems for which no elegant solutions exist: modeling asynchronous processes and coordinating events which occur at the same instant in simulated time.

The WAIT UNTIL condition statement of SOL is a modeling statement that lays out the programming requirements for implementing asynchronous

concepts. Unfortunately, these requirements are not easy to implement efficiently. For one thing, long series of tests must be made continuously to check if any WAIT UNTIL conditions have been met; these tests can slow the simulator down to such an extent that it becomes profitable not to use the asynchronous statements and program tests directly into event or activity programs. Direct scheduling timing mechanisms are generally efficient enough so that there is an overall saving in computer time if this is done.

More important than testing though, is the organizational problem of keeping track of local quantities and procedure linkages in languages that are recursive and able to organize themselves in complicated ways. To some, the benefits of asynchronous modeling do not seem worth the effort. To others, the problem is challenging and constitutes a barrier that must be overcome.

The coordination of simultaneously occurring events is another key problem. Coordination becomes important when two or more events can take place at the same instant of simulated time with each event either looking at or changing the value of a variable (or variables) common to all. Since the events must be processed sequentially, the one that "happens" first has an opportunity to change a variable's value before it is seen by other, supposedly simultaneous, events. John McNeley has proposed that a synchronous variable type be defined that only allows changes of value to occur at instants of simulation time advance. When a program attempts to change a synchronous variable value at any other time, the change is remembered, but not made. Other programs, looking at the variable see its old value. When time is

finally advanced the remembered value is substituted for the current one.

This concept almost fills the bill. It is implementable, operates on a definitional level and is easy to understand. Unfortunately it doesn't go far enough, for it does not accomodate synchronous functions. A little more work and experience will probably see this problem solved in a most satisfactory way.

The way to proceed in the design and development of timing mechanisms seems clear. The situation is quite different for data structures design. While everyone is agreed that more flexibility is needed than is presently available, the way to achieve it is not at all obvious. The concepts of ALGOL, ALGOL X, CPL, SIMSCRIPT, COBOL, PL/1, JOVIAL, LISP and almost every other language you can think of are being suggested. Proposals range from no data structures at all (everyone constructs his own) to tremendously complex definitional mechanisms for putting together predefined data structures into complex aggregates. Much of the problem stems from increasing requirements for run-time protection against semantically meaningless data accesses. With increased use being made of list processing concepts and qualifiers for data structures (subscripts, arguments) declared in program definitions such protection is necessary.

About the only thing one can say without going into different proposals in detail is that the languages of the future will contain richer data structures, will use a great deal of compile and run time analysis of data references to provide user protection, and will provide more complex data processing statements. For example, languages

will have facilities for accessing peripherally stored data automatically and efficiently. To do this they will employ asynchronous processing capabilities and support programs compiled from extensive data definitions.

New Programming Languages

It is almost impossible to talk about simulation modeling concepts and programming languages without discussing programming languages in general. For almost without exception, simulation language processors are translators to other programming languages, are themselves written in other programming languages or use compiling techniques pioneered by other programming languages. SIMSCRIPT and CSL, as we have already stated, were originally written as FORTRAN preprocessors. SIMULA will undoubtedly always be an extension to ALGOL. Today, simulation languages are being written in PL/1, and as extensions to PL/1. At one computing center in Italy they are using PL/1 to write a translator that converts source language programs to PL/1 programs. The modeling schemes we employ will always be affected by the programming languages used to implement them.

The interchange between general programming languages such as ALGOL, FORTRAN and PL/1 and simulation programming languages such as SIMSCRIPT and SIMULA is bi-directional. Many of the people involved in writing simulation languages have written algebraic compilers; their thinking and techniques have been conditioned by these experiences. Some people who have written simulation languages are involved in other compiler projects; their views on data structures and list processing facilities are being reflected in them. In addition, many of the

members of the IFIP TC.2 committee are involved in simulation language design. Much has been brought to simulation by SIMULA, whose designers and Programmers have a strong working relationship with the international ALGOL community.

There is strong pressure today to produce better simulation programming languages. Simulation users are becoming more numerous and more sophisticated; they know what facilities are available in languages such as ALGOL and PL/1, and operating systems such as OS/360. They are demanding similar facilities in simulation languages. This pressure is leading to increased use of ALGOL and PL/1 as simulation language bases. SIMSCRIPT II is one of the few new simulation programming languages not designed this way, owing no allegiance to any existing compiler, but an intellectual debt to many.

Some people, Jan Garwick of Norway for one, view this progress as alarming. They see languages becoming grotesque collections of unrelated and occasionally incompatible features, containing everything for fear they will not be able to do something. Languages are grotesque when they are hard to learn, difficult to use and grossly inefficient. True progress, these people feel, will be made when we develop simple, basic programming languages that can be used to write special purpose languages. Garwick and others are at work designing such languages.

The near future, however, is not going to see much change from what I have just described. There are too many new concepts in programming being developed for any basic language to be accepted as adequate for all purposes. Yet, since there is a magnificent exchange of ideas between language designers the languages are growing closer together.

This closeness is often not at all apparent to a casual observer, for the names applied to concepts and the mechanisms used for their implementation are often quite different. But when a language designer can look at the works of others and redefine their concepts in his own terms the future looks promising.

Hardware-Software Systems

Just as simulation languages are being affected by software advances they are being affected by advances in hardware and hardware-software systems. Every IBM 360 user feels the influence of OS/360. Time-sharing and graphics have become part of the computing vocabulary; papers are being written now on how they can be used in simulation. Some work has already been done which is more than research in nature: engineers at Boeing Huntsville have been using a graphs-oriented simulation system for some time employing GPSS and the IBM ALPINE system. Project MAC has had OPS-3, an interactive simulation programming system, in use since 1965 learning how to use time-sharing effectively to build models and perform simulation experiments.

Some simulation languages have a graphics capability today. Both GPSS/360 and GSP contain statements that generate hard copy graphic displays. But no existing languages have the interactive graphical capability we would like to have. This is not surprising, for when we discuss such facilities we talk about a world which is still available to very few. Most language designers have been working with non-time-shared computers without graphical devices. They have been addressing themselves to problems of formulating models, not to problems

of using them. When the dust settles and some of the pressing modeling questions are solved, attention will be directed to time-sharing and graphics.

The future will see a great deal of attention paid to interactive modeling, both graphical and narrative. Work has barely begun on graphical languages that are suitable for expressing simulation concepts. Some papers have been written but few, if any, usable systems have been produced or reported on. The intuitive appeal of graphical modeling is great. As with all things the soundness of the idea will have to be tested in the field. There are human engineering and psychological aspects to graphical modeling that will pose new problems, problems not thought of by the technicians who design the modeling systems. The number of separate relationships a person can comprehend on a CRT display and juggle in his mind will be an important factor. Perhaps the size of display needed to make the technique worthwhile will be beyond the reach of today's technology or pocketbook for a long while. A well directed project or two should be able to answer some of the more important questions in the next year or two.

Graphics and time-sharing also have a large future in simulation experimentation. While it is true that many simulation jobs are well structured and involve little more than substantial computer runs to estimate system properties, a large number of people have jobs that are unstructured. They are interested in searching for parameter settings that yield optimum system performance or want to learn something about the shape of a system's response surface. Batch-type processing hinders rather than helps these jobs. A few minutes of

console interaction with the right kind of displays can probably displace hours of conventional hit and miss experimentation. Little work has been done on this aspect of interactive simulation. It seems like a very profitable area.

THE FUTURE

Most of the work done today in simulation is in computer language development. In addition to designing modeling languages, simulation researchers have developed compilers, worked on varieties of source language translators and learned to live with the almost limitless amount of detail necessary to get a computer language in working order. Much creative energy has been channeled in areas other than that of modeling language development. Many of these diversions have been challenging and have produced fruitful results; many others have simply expanded the time required to complete a project.

Current research on compiler-compilers leads one to be hopeful that these days are past, that the day is near when a person can write the syntax and semantics of a modeling language and have a compiler for that language produced for him. When this day arrives, experimentation at the modeling concept level will be more rapid and will produce improved modeling concepts at a greatly accelerated rate. Users will no longer be forced to choose between concepts, be they those of timing mechanisms, data structures or what ever. It will be possible to tailor a compiler to individual problem area needs, incorporating as many concepts of whatever nature are needed to do an efficient job. For a change the tail will not wag the dog.

Simulation languages, if we can even call these generated compilers by that name, will also be simulation systems. They will contain facilities for running simulation experiments, for statistically deciding how long to run a model and for optimizing system performance by automatic parameter adjustment. When time-shared, they will be interactive and, when possible, accept graphic input. They will be self-instructive, powerful and easy to use.

Let us make sure that we use them correctly, on the right problems and to the right ends.

BIBLIOGRAPHY

1. Bennet, R. P., Cooley, P. R., Hovey, S. W., Kribs, C. A. and Lackner, R., "SIMPAC Users Manual," TM-602/000/00, Systems Development Corporation, April 1962.

2. Bibliography on Simulation, The IBM Corporation, 320-0924-0, 1966.

3. Blunden, G. P. and H. S. Krasnow, "The Process Concept as a Basis for Simulation Modeling," IBM ASDD Technical Report 17-181, November 1965.

4. Brennan, R. D. and R. N. Linebarger, "An Evaluation of Digital Analog Simulator Languages," Simulation, Vol. 3, No. 6, December 1964.

5. Buxton, J. N., and J. G. Laski, "Control and Simulation Language," The Computer Journal, Vol. 5, No. 3, 1962.

6. Clancy, J. J. and Mark S. Fineberg, "Digital Simulation Languages: A Critique and a Guide," Proceedings of the Fall Joint Computer Conference, 1965.

7. Control and Simulation Language Users Manual, IBM United Kingdom Limited Data Centre, 1966.

8. Dahl, O. J., and K. Nygaard, "The SIMULA Language," Report from the Norwegian Computing Center, May 1965.

9. Extended C.S.L., Honeywell Controls Limited, Electronic Data Processing Division, Great West Road, Brentford, Middlesex, England.

10. Famolari, E., "FORSIM IV User's Guide," SR-99, The MITRE Corporation, February 1964.

11. Farmer, J. A. and R. H. Collcutt, "Experience of Digital Simulations in a Large O. R. Group," BISRA Open Report, OR/CA/38/65, September 1965.

12. Frye, W. H., The Interrelationship Graph (IRG) as a Technique for Computer Model Development, Navy Electronics Laboratory, SF 001 0102, Task 11276 (NEL Z 21761), November 1965.

13. Ginsberg, A. S., H. M. Markowitz, and Paula M. Oldfather, "Programming by Questionnaire," RM-4460-PR, The RAND Corporation, April 1965.

14. Gordon, G., "A General Purpose Systems Simulator," IBM Systems Journal, September 1962.

15. Greenberger, M., M. M. Jones, J. H. Morris, Jr., and David N. Ness, On-Line Computation and Simulation OPS-3, The MIT Press, 1965.

16. Hammersly, J. M., and D. C. Handscomb, Monte Carlo Methods, Methuen & Co., Ltd., London, 1964.

17. Herscovitch, H. and T. H. Schneider, "GPSS III - An Expanded General Purpose Simulator," IBM Systems Journal, Vol. 4, No. 3, 1965.

18. Job Shop Simulation Application, IBM Corporation Data Systems Division, M & A-1, 1960.

19. Karr, H. W., H. Kleine, and H. M. Markowitz, "SIMSCRIPT I.5," C.A.C.I.65-INT-1, California Analysis Center, Inc., June 1965.

20. Kelley, D. H., and J. N. Buxton, "Montecode - An Interpretive Program for Monte Carlo Simulations," Computer Journal, Vol. 5 No. 2, 1962.

21. Kiviat, P. J., "GASP - A General Activity Simulation Program," P-2864, The RAND Corporation, 1964.

22. Knuth, D. C., and J. L. McNeley, "SOL-A Symbolic Language for General-Purpose Systems Simulation," IEEE Transactions on Electronic Computers, August 1964.

23. Krasnow, H. S. and R. A. Merikallio, "The Past, Present and Future of General Simulation Languages," Management Science, Vol. 11 No. 2, 1964.

24. Krasnow, H. W., "Dynamic Representation in Discrete Interaction Simulation Languages," IBM ASDD Technical Report 17-171, August 1965.

25. Lackner, M. R. and Patricia Kribs, "Introduction to a Calculus of Change," TM-1750/000/01, Systems Development Corporation, February 1964.

26. Lackner, Michael R., "Conversational Modeling and Simulation," in Proceedings of the IFIP Congress 65, Washington D.C., Spartan, 1965.

27. Laski, J. G., "On Time Structure in (Monte Carlo) Simulations," Operational Research Quarterly, Vol. 16, No. 3, 1965.

28. Markowitz, H. M., B. Hausner and H. W. Karr, SIMSCRIPT: A Simulation Programming Language, "Prentice-Hall, Inc. 1963.

29. MILITRAN Programming Manual, Report No. ESD-TDR-64-320, Systems Research Group, Inc., June 1964.

30. Parente, R. J., "A Language for Dynamic System Description," IBM ASDD Technical Report 17-180, 1965.

31. SHARE Systems Simulation Project Newsletter, March 1966.

32. Shubik, Martin, "Bibliography on Simulation, Gaming, Artificial Intelligence and Allied Topics," J. Am. Stat. Assoc., Vol. 55, December 1960.

33. SIMCOM User's Guide, TR-65-2-149010, General Electric Company, Information Systems Operations, 1964.

34. Teichrow, D. and John F. Lubin, "Computer Simulation: Discussion of the Technique and Comparison of Languages," Graduate School of Business, Working Paper No. 20, Stanford University, August 1964.

35. Tocher, K. D., "Review of Simulation Languages," Operational Research Quarterly, Vol. 16, No. 2, June 1965.

36. Tocher, K. D., and D. A. Hopkins, "Handbook of the General Simulation Program, Mk. II," The United Steel Companies, Ltd., Department of Operational Research, Report No. 118/ORD 10/TECH, 1964.

37. Tocher, K. D., "Handbook of the General Simulation Program. Volume I," The United Steel Companies Limited, Report No. 77/ORC 10/TECH 1961.

38. Workshop on Simulation Languages, University of Pennsylvania, Management Science Center, March 17-18, 1966.