Proceedings of the 1984
Winter Simulation Conference
S. Sheppard, U. Pooch, D. Pegden (eds.)

DISCRETE EVENT SIMULATION LANGUAGES

CURRENT STATUS AND FUTURE DIRECTIONS

James O. Henriksen
Wolverine Software Corporation
7630 Little River Turnpike - Suite 208
Annandale, VA 22003-2653

ABSTRACT

Simulation software is changing. Within the past several years, significant developments in simulation software have taken place:

1. New simulation languages have been developed.
2. New software packages have been developed for use in conjunction with simulation (for purposes other than building models, per se.)
3. New features have been added to existing languages.
4. Vendors new to the simulation community have marketed implementations of existing software packages.
5. Simulation environments, comprising integrated collections of simulation software tools have been built.

As a consequence of these developments, those readers whose perceptions of simulation software are several years old should consider themselves out of date. Those readers whose perceptions are five or more years old should consider themselves extremely out of date. Furthermore, enormous amounts of time and energy are presently being expended on research and development of simulation software. Thus, we can expect dramatic changes to take place in the near future. Simulation software of the 1990's will be as far removed from present software as present software is removed from building models "from scratch" in languages such as Fortran.

This paper, a tutorial, summarizes the present state of simulation software, identifies pressures for changes, and describes an emerging consensus on the major characteristics of simulation software of the future.

DISCLAIMER

This paper is not intended to be an exhaustive survey. Due to a focus on directly software-related issues and due to space limitations, many important contributions to the current state of the art of simulation, and some opinions about directions of future growth have necessarily been omitted.

THE NATURE OF THE PROBLEM

Size and Complexity

The development of simulation software tools is an enormous problem. The disparate nature of the component disciplines comprising simulation guarantees that the progress of the science (art?) of simulation will be less orderly than we would prefer. Economic considerations amplify the tendency toward disorderly progress. Assume, for example, that a salesman, an engineer, and a computer scientist must compete for resources within a company. The salesman can argue that money spent in support of his activities (acquisition of fancy color graphics hardware and software, for example) correlates directly with sales potential. The engineer argues that money spent in support of his activities allow him to design and build better products. The computer scientist argues that money spent in support of his activities allows him to build theoretical and pragmatic (if absolutely necessary) foundations under everyone else. Guess who gets the money.

Simulations are examples of what have been aptly termed "domain-dependent" [1] problems. In the process of building and experimenting with a simulation model, our concepts about the system we are modeling almost always undergo significant change. Simulation software must recognize the implied need for feedback mechanisms among the experimentation, implementation, and design areas of a simulation project. Such feedback must not only be allowed; it must be encouraged [2]. Unfortunately, software development methodologies typically do not encourage such feedback; if fact some discourage it. (See the section entitled The Evils of Structured Programming, below.)

Nuts and Bolts

To put simulation software into perspective, it is useful to first consider a smaller problem, the development of tools for "ordinary programming." The present state of the art of programming tools leaves a great deal to be desired. Gutz, et al. [3] lament "the primitive state of the programmer's art" and criticize current "primitive software tools" for "lack of compatibility, ill-defined capabilities, lack of uniformity, lack of tailorability, lack of support of software evolution, and lack of evaluative data." To rectify this situation, enormous amounts of time and energy are being expended on research into software development tools and methodologies. (See reference [4] for an excellent bibliography.) Languages such as Modula 2 and ADA represent early milestones on the path to improved programming productivity.

To many, programming is the most obvious activity in performing a simulation; however, in the overall scheme of things, programming must be regarded as only a component of a larger discipline. Designing and building models is a much more difficult undertaking than designing and building programs. Preparation of model input, specification of model design, and analysis of model output require skills beyond those required for "mere" programming. Programming is emphasized in this paper, because our primary concern herein is with simulation languages. The emphasis on programming is also justifiable on a philosophical plane. Scientific progress almost always proceeds from the concrete to the abstract. Theories are

devised to explain observed phenomena, and experiments are devised to test theories. Generality of understanding is the result of scientic progress, not the starting point. By its very nature, simulation is an experimental science. Thus, we have every reason to expect that advances in programming will play a primary role in advancing the science of simulation as a whole.

Simulation software tools exist for other activities in addition to programming. For example, software is used for fitting raw data to statistical distributions, portraying model operation (e.g., animation), displaying model outputs, and analyzing model outputs. All of these tools operate at a "nuts and bolts" level. The lessons learned in the construction and application of these tools will help to build stronger underlying methodologies.

Our emphasis on "nuts and bolts" issues should not be construed as discounting the importance of "pure" research into the more theoretical aspects of simulation. We agree with Tuncer Oren's description (in the foreword to Zeigler's Multifacetted Modelling [5]: "Model-based simulation is like a gem: it is multifacetted. Some of the specialists too close to one of the facets, perceive only that single facet and the reflection of the success of their careers through it." "Nuts and bolts" software issues are an important facet of simulation. Unfortunately, however, their importance relative to other issues tends to be overestimated by simulation practitioners and underestimated by theoreticians.

Simulation Environments

An integrated simulation environment [6] is a collection of software tools for designing, writing, and validating models; writing and verifying simulation programs (implementing models); preparing model input data; analyzing model output data; and designing and carrying out experiments with models. The environment includes mechanisms for feedback from the experimentation stage to the implementation stage to the design stage. To be worthy of the adjective integrated, simulation systems must allow the modeller to shift his attention among various problem areas with minimal difficulty. For example, in an integrated simulation environment, a user might choose to momentarily suspend the execution of a model, to browse through model design specifications and/or source code, before resuming execution. With most present software systems, shifting one's attention among problem areas is cumbersome. Many simulation languages do not provide for interactive control of model execution; thus suspension of execution is impossible. Of those which do, very few allow the user to edit the source code or examine design specifications. Typically, the model design tools (if any) and source code editor can only be invoked by operating system-level commands; i.e., model execution, editing, and design are mutually exclusive activities.

Improvements on Multiple Levels

Improvements in simulation software will take place on three levels. At the lowest level, progress will be made in refining underlying theories and methodologies. At the middle level, progress will be made in specific problem areas, such as model design, programming, preparation of input, and presentation and analysis of output. In some cases, old tools will be improved, but in others, new tools will have to be developed. At the highest level, progress will be made in integrating simulation software tools into simulation environments.

Summary

To summarize the situation, simulation is in a state where:

1. The domain-dependent character of simulation applications imposes requirements which transcend those of "ordinary" programming systems;
2. Major component disciplines, e.g., programming, are in need of improvements;
3. To reach maturity as a science, simulation must make progress on multiple levels;
4. For philosophical and economic reasons, progress can be guaranteed to be undesirably disorderly.

Is this cause for pessimism? No, no, no! Great progress has been and is being made. Many contributions to the progress will come from within the simulation community, but many other ideas will be borrowed from other disciplines, such as artificial intelligence. Optimism is in order.

DEVELOPMENTS WITHIN THE SIMULATION COMMUNITY

The Impact of Advances in Hardware

Hardware capabilities are increasing and costs are decreasing on a daily basis. The comparative importance of labor costs versus equipment costs is being restored to a proper balance, where people are regarded as a more precious resource than machines. Software vendors have responded to this situation by implementing software systems for use on low-cost, desktop hardware. Time-tested, major languages such as GPSS [7,8], SLAM [9], and Simscript II.5 [10] have all been implemented for desktop systems, as have languages such as SIMAN [11].

The dominant role of mainframe computers in simulation has been successfully challenged by smaller, easier to use systems. What a desktop system may lack in horsepower, it makes up through greatly improved ease of use. The user of a desktop system needs no staff of operators to run his machine. The system is "up" whenever he turns it on, and response times are extremely predictable. Perhaps best of all, there's no monthly bill from the accounting department. Many simulation practitioners will continue to have need of mainframe computing horsepower, but they too will benefit from desktop technology, by using the desktop machine to develop and experiment with modeling concepts, and using the mainframe to make production runs. To operate in this manner, one must have the ability to upload and download programs and data, and mainframe and desktop software systems must be completely compatible.

New Software Tools

New software tools are available in a variety of simulation-related areas. Examples include:

1. Preprocessor software [12], which allows a user to build simulation models out of building blocks which operate at a higher level than traditional simulation languages.
2. On-line and post-processing graphics software [13], which facilitates the portrayal of model operation and display of model results.

3. Statistical tools [14,15], facilitating the fitting of distributions to data and the analysis of simulation outputs.
4. Database tools [16], providing an underlying, common repository for simulation inputs and outputs.
5. Model design languages [17], providing partially automated assistance in the design and documentation of models.

Integrated Simulation Environments

The TESS system is the first integrated simulation environment to be offered commercially. It "represents a new generation of software that integrates model building, simulation execution, and the analysis and presentation of results [18]." The system "has evolved over a ten-year period to provide support for problem solving using simulation." For the most part, TESS is a non-procedural language; i.e., users express what is to be done, and TESS itself determines how such requests are to be carried out. TESS provides a collection of independent subsystems; e.g., the design of a model, the specification of run controls, and the preparation and specification of input data for particular experiments can all be performed independently. However, the subsystems are integrated into an overall conceptual framework. For example, the combination of a model and its run controls and input data constitute what is called a scenario. Scenarios are stored in a database. When a simulation is run according to the specifications of a scenario, results are recorded in the database. Reports, graphic displays, and animations can be generated, as required, by issuing non-procedural requests to appropriate subsystems. If necessary, multiple passes can be made over the same data, to refine the quality of output presentation.

Pressure for Language Improvements

GPSS provides an interesting example of pressures for fundamental, low-level changes to simulation languages. These days, GPSS models frequently contain 5,000 or more statements, and they may take months to build. (This is an interesting contrast to the expectations of Geoffrey Gordon, the inventor of GPSS, who anticipated that typical problems to which the language would be applied would result in 200-300 statement models built over 1-2 weeks' time.) The GPSS language has no provisions for separately compiled modules (except for HELP and EXTERNAL routines written in other languages). A 5,000-statement program is B-I-G. Given that models of ever-increasing size and complexity are being built, will the typical GPSS program of tomorrow contain 25,000 statements? While a 5,000-statement program may be cumbersome, a 25,000-statement program is bound to be extremely unwieldy. To cope with such problems, the GPSS of tomorrow will have to be radically different from the GPSS of today. Forthcoming papers [19,20] will describe a system designed in response to such pressures.

DEVELOPMENTS OUTSIDE THE SIMULATION COMMUNITY

In this section, we explore developments which are taking place outside the simulation community, but which will also contribute to simulation. Improvements borrowed from other disciplines, as we shall see, constitute something less than a free lunch.

The Evils of Structured Programming

Over the past fifteen years, a body of techniques collectively referred to as structured programming has been developed. Structured programming means different things to different people, but sufficient commonality exists to allow one to speak meaningfully of structured programming methodology. The virtues of structured programming have been extolled ad nauseum (for a representative collection, see reference [21]). For all its virtues, structured programming has two major defects: it overemphasizes the concept of iterative refinement (a process of synthesis), and it underemphasizes the concept of feedback (a process of analysis).

The heart of structured programming is the process of iterative refinement, in which one starts with an abstract characterization of system requirements and proceeds through a series of steps in which implementation details are added, rendering abstract specifications more concrete. Great emphasis is placed on assuring the correctness of the individual transformations from the abstract to the concrete. Unless great care is exercised, this process may imply a methodological rigidity which is particularly inappropriate to simulation. In a classic, if somewhat pessimistic commentary, Sheil [22] states:

> Virtually all modern programming methodology is predicated on the assumption that a programming project is fundamentally a problem of implementation. The design is supposed to be decided upon first, based on specifications provided by the client; the implementation follows. The dichotomy is so important that it is a standard practice to recognize that a client may have only a partial understanding of his needs, so that extensive consultations may be required to ensure a complete specification with which the client will be happy. This dialog guarantees a fixed specification that will form a stable base for an implementation.
>
> The vast bulk of existing programming practice and technology, such as structured design methodology, is designed to ensure that the implementation does, in fact, follow the specification in controlled fashion, rather than wander off in some unpredictable direction. And for good reason. Modern programming methodology is a significant achievement that has played a major role in preventing the kind of implementation disasters that often befell large programming projects in the 1960s.
>
> The implementation disasters of the 1960s, however, are slowly being succeeded by the design disasters of the 1980s...

Sheil argues quite persuasively that as the size and complexity of programming projects increases, the process of iterative refinement is less and less likely to be successful. To escape the straitjacket of iterative refinement, we must recognize that downstream discovery of less than perfect, or even erroneous, assumptions is something to be expected in complex systems. We need software systems which not only allow for, but encourage feedback from "later" stages to "earlier" stages.

An overcommitment to iterative refinement may result in an undercommitment to analytical skills. The author once attended a professional development seminar given by Edsger Dijkstra, regarded by many as

the father of structured programming. By sheer happenstance, I ended up sitting next to Dijkstra at lunch. I asked him what he would do when his techniques for building correct programs failed; i.e., what would he do if he made a mistake and had a difficult bug in a program. His flippant response was "I'd consult an expert." He never answered the question. Dijkstra is not alone in downplaying the importance of analytical skills. Niklaus Wirth, the inventor of Pascal and Modula 2, states [23] that "Instead of relying too much on antiquated 'debugging tools' or futuristic automatic program verifiers, we should give more emphasis to systematic construction [emphasis added] of programs and languages that facilitate transparent formulation and automatic consistency checks." Those who blindly overemphasize the synthetic nature of programming at the expense of analytical skills will sooner or later have to rely on someone who doesn't: the intractable bug may be just around the corner.

Improvements to Structured Programming Methodology

Recently, increased attention has been given to the relationship between software and the environment in which it operates. Giddings [1] defines the notion of domain-dependent software, in which "The development process is embedded within a search for knowledge about the domain of discourse." A simulation program is a classic example of domain-dependent software. It is encouraging to see an article such as Giddings' published outside the simulation community, because it indicates a very favorable direction for future research efforts. It is interesting to compare Giddings' guidelines for domain-dependent software with Nance's Conical Methodology [2], which was developed specifically for handling simulation problems.

Improved Mechanisms for Abstraction

Abstraction mechanisms provide a means whereby a program can use (read/write) attributes of objects without knowledge of how the attributes are implemented, and can implement objects without knowledge of how they are to be used. The oldest and still perhaps best known abstraction mechanism is the class concept of Simula [24]. The most common form of abstraction is the concept of an abstract data type. With an abstract data type, information about the implementation of data is hidden from the user of the data; information is made available only through very carefully controlled interfaces. For example, an operating system might have a "next task to be executed" attribute which was referenced from a number of locations. To implement this attribute as an abstract data type, one would provide procedures to determine and to alter the next task to be executed. If the operating system were implemented in this manner, alternative algorithms for handling task dispatching (FIFO, priority queues, etc.) could easily be substituted for one another.

Other forms of abstractions include procedural abstractions, in which procedures convert collections of inputs into collections of outputs (e.g., pipes in Unix [25]), and control abstractions, which define methods for sequencing arbitrary actions (e.g., the FOR EACH [member of a set] construct of Simscript II.5). Languages which have built upon Simula's class concept include CLU [26], Alphard [27], Modula 2 [28], and ADA [29]. The applicability of abstraction mechanisms to simulation is obvious, for abstraction is the most essential property of a model. Simulation languages of the future will benefit from the lessons learned in the development and use of these mechanisms.

Extensible Languages

An extensible language provides mechanisms for adding new operators and data types to the language. One of the earliest extensible languages was MAD [30]. MAD provided a simple, but powerful operator definition facility. Statements defining operators were translated into the same tabular form used by the compiler to represent built-in operators. The lowest level code generation routines did not distinguish between user-defined operators and predefined operators. Using the operator definition facility, users could add new data types to the language, add new operators, map new operators into previously defined operators, and add support of new operand modes to previously defined operators. The operator definition facility was used to define a number of sizable "packages." For example, packages were developed for matrix arithmetic, complex arithmetic, double precision arithmetic, and string manipulation.

A limited subset of the MAD operator definition concepts have resurfaced in ADA. ADA operator overloading allows the user to add support of new operand mode combinations to existing operators. For example, the ADA base language does not permit mixed-mode addition; i.e., a real number cannot be added to an integer. However, the addition operator can be overloaded by the user, to allow such operations.

An interesting alternative approach to extensibility is the use of source macros, as provided in the C language [31]. In many cases, a good source macro capability can approximate the power of an operator definition facility, although macros are not nearly as elegant. As an example, assume we are building a model of a computer system, using a time unit of seconds, and we would like to add the capability to easily specify times in milliseconds. In a language like C, we might be tempted to use a simple text-substitution macro, such as

```
#define MILLISECONDS  * 0.001
```

Thus, a statement of the form

```
wait 10 MILLISECONDS;
```

would be expanded into

```
wait 10 * 0.001;
```

by the compiler. Unfortunately, a statement of the form

```
wait i + j MILLISECONDS;
```

would be expanded into

```
wait i + j * 0.001;
```

Thus, we would fail to achieve the desired effect of scaling the entire expresion, "i+j". To get around this problem, we could define the following C macro:

```
#define MILLISECONDS(time) (time) * 0.001
```

A statement of the form

```
wait MILLISECONDS(i+j);
```

would be expanded into

```
wait (i+j) * 0.001;
```

While this would achieve the desired effect, the notation employed is downright ugly. An operator definition capability would allow use of the natural notation first shown above, and it would enforce the expected operator precedence. A definition for MILLISECONDS might look as follows:

```
define right unary operator MILLISECONDS(xpr)
      xpr * 0.001
```

Formalizing Modularity

It is generally useful to structure a large program as a collection of modules. Most programming languages do not include rigorous definitions of what constitutes a module. Some languages, such as GPSS, do not even provide for independent compilation of modules. While useful standards can be developed for modularity in languages lacking formal definitions of modules, languages which do include formally defined modules offer a distinct advantage. Languages such as Simula, CLU, Alphard, Modula 2, and ADA provide such mechanisms in varying degrees. Simula is the only one of these languages specifically designed for use in simulation. Due to the increasing size and complexity of simulation applications, simulation languages of the future will have to a better job of supporting formally defined modules.

The UNIX operating system provides a utility called "make," which is used for manipulating collections of modules. Make operates on a user-supplied "makefile" which describes the relationships among files comprising a large program. (Files can be thought of as an informal approximation to modules.) Relationships describe actions to be taken when a file has been changed. A typical rule might specify that when file X has been changed, files A, B, and C should be recompiled. When using the make utilty, one simply types "make" after making arbitrary changes to an arbitrary number of files. The make utility detects instances of file changes and takes all necessary actions to build a consistent new version of the entire program.

The make utility is (most unfortunately) completely dependent on the accuracy of the makefile, which is manually constructed by the user. In an ideal world, the make utility would interface with language compilers and editors. Language compilers could automatically deduce dependencies not only at the file level, but at a much lower level of detail. Similarly, editors could communicate to the make utility the details of changes made to a file. For example, if a subroutine definition were altered by changing the mode of one of its arguments, then files containing invocations of the subroutine would have to be edited and recompiled to achieve a consistent system. Integrated software environments of the future will provide such support.

Artificial Intelligence

The tools being developed for dealing with artificial intelligence will benefit simulation in several major ways. First, AI researchers are working on the all-important problem of knowledge representation. The concept of a knowledge base will be essential to simulation software of the future. We don't want to reinvent the wheel forever; what we'd like to be able to do is to sit down at a terminal and say "I want to do a new simulation which is like the one I did yesterday, except..." Someday, dialog of this sort will be possible between man and machine. Second, AI researchers are accustomed to dealing with ill-formed, incompletely specified problems. Lisp [32], the most popular language in the AI community, reflects the character of such problems: Lisp programs are very dynamic: identifiers and their meanings can be determined at run time, programs can generate and execute other programs, etc. While this degree of flexibility may not be necessary for typical simulation applications, it provides an interesting reference point in the spectrum of programming languages.

THE FUTURE

These are exciting times to be involved with simulation software. New, low-cost hardware has put the computational power of yesterday's mainframe onto today's desktop. Old software is being enhanced and retargeted for use on such machines, and new software tools are being developed. Contributions are being made by members of the simulation community, and in the future, judicious borrowing of ideas developed within other disciplines will have a great impact. Only a few years ago, people were writing about simulation environments [6]. Now the first of the environments is a commercially available tool. All of these developments are indications of great things yet to come.

Since there are many groups working on simulation software, it is impossible to predict the exact nature of simulation software of the future; however, there is an emerging consensus among developers as to the general direction in which their research is headed. Among the generally agreed upon attributes are the following:

1. Simulation software will be implemented for use on powerful desktop single-user machines.
2. It will be possible to easily transfer programs and data among machines: networks will become commonplace.
3. Software tools will be consolidated into integrated environments.
4. Languages will provide better mechanisms for abstraction.
5. Languages will support extensibility.
6. Languages will be improved to incorporate formal definitions of modules, and software tools will be developed to support manipulation of modules. Ultimately, such tools will interface with model design tools, to provide assistance in verifying that a simulation program is faithful to its formal design.

BIBLIOGRAPHY

1. Giddings, R. V., "Accommodating Uncertainty in Software Design," Communications of the ACM, Volume 27, Number 5, pp. 428-434, May, 1984.

2. Nance, R. E., Technical Report CS81003-R: "Model Representation in Discrete Event Simulation: The Conical Methodology," Computer Science Department, Virginia Polytechnic Institute and State University, Blacksburg, 1981, 71 pp.

3. Gutz, S., Wasserman, A. I., and Spier, M. J., "Personal Development Systems for the Professional Programmer," IEEE Computer Magazine, pp. 45-53, April, 1981.

4. Wasserman, A. I., and Gutz, S., "The Future of Programming," Communications of the ACM, Volume 25, Number 3, pp. 196-206, March, 1982.

5. Zeigler, B. P., Multifacetted Modelling and Discrete Event Simulation, Academic Press Inc., Orlando, 1984, 372 pp.

6. Henriksen, J. O., "The Integrated Simulation Environment," Operations Research, Volume 31, Number 6, pp. 1053-1073, November-December, 1983.

7. GPSS/PC User Manual, Minuteman Software, Stow, MA, 1984.

8. Henriksen, J. O., and Crain, R. C., GPSS/H User's Manual, Second Edition, Wolverine Software Corporation, Annandale, VA, 1983.

9. Pritsker, A. A. B. , Introduction to Simulation and SLAM II, Second Edition, Systems Publishing Corporation, West Lafayette, 1984.

10. Russell, E. C., Building Simulation Models with SIMSCRIPT II.5, C.A.C.I., Los Angeles, 1983.

11. SIMAN product literature available from: Systems Modeling Corporation, P.O. Box 10074, State College, PA 16805.

12. AutoMod product literature available from: AutoSimulations, Inc., P.O. Box 633, Bountiful, UT 84010.

13. AutoGram product literature available from: AutoSimulations, Inc., P.O. Box 633, Bountiful, UT 84010.

14. Unifit product literature available from: Simulation Modeling & Analysis Co., P.O. Box 40996, Tucson, AZ 85717.

15. AID product literature available from: Pritsker & Associates, Inc., P.O. Box 2413, West Lafayette, IN 47906.

16. Standridge, C. R., "An Introduction to the Simulation Data Language," Proceedings of the 1981 Winter Simulation Conference, 1981, pp. 635-637.

17. Kleine, H., Software Design and Documentation Language, JPL Publication 77-24, Revision 1, NASA Jet Propulsion Laboratory, Pasadena, 1977.

18. Standridge, C. R. et al., Performing Simulation Projects with TESS, Pritsker & Associates, Inc., West Lafayette, 1984.

19. Henriksen, J. O. "Introducing GPSS/85", To be presented at Eighteenth Annual Simulation Symposium, Tampa, FL, March, 1985.

20. Henriksen, J. O. "An Extensible Host Language for Simulation Environments", To be presented at 11th IMACS World Congress, Oslo, Norway, August, 1985.

21. Yourdon, E. N. , ed., Classics in Software Engineering, Yourdon Press, New York, 1979.

22. Sheil, B. A. , "Power Tools for Programmers," Interactive Programming Environments, McGraw-Hill, New York, 1984.

23. Wirth, N., "An Assessment of the Programming Language Pascal," IEEE Transactions on Software Engineering, pp. 192-198, June, 1975.

24. Franta, W. R. , The Process View of Simulation, North-Holland, New York, 1977, 241 pp.

25. Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System", Communications of the ACM, Volume 17, Number 7, pp. 365-375, July, 1974, reprinted in Communications of the ACM, Volume 26, Number 1, pp. 84-89, January, 1983.

26. Liskov, B., et al., "Abstraction Mechanisms in CLU," Communications of the ACM, pp.564-576, August, 1977.

27. Shaw, M. and Wulf, W. A., "Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators," Communications of the ACM, pp. 553-564, August, 1977

28. Wirth, N., Programming in Modula-2, Second Edition, Springer-Verlag, Berlin, 1982.

29. Olsen, E. W. and Whitehill, S. B., Ada for Programmers, Reston Publishing Company, Inc., Reston, 1983.

30. The Michigan Algorithm Decoder (The MAD Manual), Revised Edition, 1966 (Out of print).

31. Kernighan, B. W. and Ritchie, D. M., The C Programming Language, Prentice-Hall, Inc., Englewood Cliffs, 1978.

32. Charniak, E., Riesbeck, C., and McDermott, D., Artificial Intelligence Programming, Lawrence Erlbaum Associates, Hillsdale, NJ, 1980.