Proceedings of the 1993 Winter Simulation Conference
G.W. Evans, M. Mollaghasemi, E.C. Russell, W.E. Biles (eds.)

SLX, THE SUCCESSOR TO GPSS/H

James O. Henriksen
Wolverine Software Corporation
4115 Annandale Road
Annandale, VA 22003-2500

ABSTRACT

This paper describes SLX, Wolverine Software's "next generation" successor to GPSS/H. SLX retains fundamental concepts of GPSS/H, such as its Transaction-flow world-view, Facilities, Queues, and Storages. However, SLX replaces some GPSS/H features entirely, and it replaces collections of disparate GPSS/H features with simpler, more general features.

SLX is much more than a new implementation of GPSS/H. SLX is a layered modeling system in which GPSS/H comprises but one of the layers. SLX provides a foundation under GPSS/H and powerful extensibility mechanisms which facilitate the development of higher level, application-specific modeling tools.

The sections which follow provide an overview of SLX, and describe the goals of the development of SLX, the architecture of SLX, the SLX environment, concepts retained from GPSS/H, features modeled after other languages, and the extensibility mechanisms of SLX. The relationship of SLX to other modeling tools is discussed in Section 8, entitled "Can Do, Can't Do, Must Do, Should Do, Shouldn't Do."

Since SLX is still under development as of this writing, this paper emphasizes design philosophies and principles, rather than concrete, syntactically accurate examples. The presentation of this paper will include a "live" demonstration.

1 OVERVIEW

SLX is a layered modeling system; its layers are shown in Figure 1.

Level 4:	Packages for the Non-Simulationist
Level 3:	Application-Specific Packages
Level 2:	Next-Generation GPSS/H
Level 1:	Simulation / Statistical Primitives
Level 0:	Kernel (Largely Modeled after C)

Figure 1: The Layers Comprising SLX

Level 0, the SLX kernel, is a language loosely modeled after the C language, and a run-time support library. The kernel language is compiled directly into machine instructions for the Intel 386/486/Pentium chip family. The kernel provides support for a number of primitives required for simulation. For example, simulation "wait until" and statistics collection features rely on kernel support for detecting when changes are made to a monitored variable.

Level 1 consists of data structures, subroutines, operators, and macros, all written in SLX. These features provide simulation support for higher levels of SLX. An SLX user can augment level 1 by adding similar capabilities of his or her own design.

Level 2 is the "new" GPSS/H. It retains many of the basic concepts of GPSS/H, but implements them in a more general way. The computational portions of GPSS/H (Savevalues, Matrix Savevalues, Ampervariables, etc.) have been supplanted by level 0 and level 1 SLX capabilities.

Level 3 is the level at which application-specific packages will be developed, e.g., manufacturing, health care, telecommunications. Wolverine Software will develop some of these packages, but we anticipate that others will be developed by third parties.

Level 4 is the level at which very high level packages will be developed for use by non-simulationists. These packages will feature graphically-based, highly interactive front ends.

SLX was designed to capture the strengths of GPSS/H. The basic concepts retained from GPSS/H are discussed in Section 4, below. While SLX captures the strengths of GPSS/H, its layered approach and extensibility mechanisms provide it with a stronger foundation and greater adaptability to end-user applications. Users are encouraged to use the highest layers of SLX, since they are the most powerful. However, when a given layer does not contain a required feature, users have recourse to lower-level mechanisms. For example, if a user wishes to add a new statement to the GPSS/H-level of SLX, (s)he has access to the same mechanisms we used to implement the built-in statements. This is a hallmark of SLX's open architecture.

Perhaps the greatest strength of SLX lies in its novel extensibility mechanisms for getting from layer to layer. (Just having layers is no good if it's too difficult

to get from layer to layer.) While some of these mechanisms are borrowed from the object-oriented paradigm (OOP), many are new developments. Extensibility mechanisms are discussed in further detail in Section 7, below.

2 GOALS IN THE DEVELOPMENT OF SLX

The most important goal in the development of SLX was to retain the strengths of GPSS/H. Our motivations for doing so were threefold. First, we really believed in the strengths of GPSS/H. GPSS/H has been in use for over fifteen years, its longevity attributable to ease-of-use, ease-of-learning, power, flexibility, and superior performance. Second, we wanted to provide an evolutionary, rather than revolutionary, growth path for existing users of GPSS/H. Finally, we had over thirty man-years' experience implementing GPSS/H in a group of senior developers, and even more experience in dealing with users of GPSS/H. Retaining the strengths of GPSS/H allowed us to capitalize on our strengths as language developers.

A second goal in the development of SLX was to remove outdated GPSS/H features and, wherever possible, to replace collections of disparate features with simpler, more general features. For example, in GPSS/H, there are at least eight different forms of the assignment statement: ASSIGN, SAVEVALUE, MSAVEVALUE, LOGIC, LET, BLET, INDEX, and INITIAL. Each of these statements has its own syntax and semantics. In some cases, the syntax and semantics are peculiar and non-obvious. SLX replaces these statements with assignment statements and variable initialization whose syntax is modeled after the C language.

A third goal in the development of SLX was to generalize the implementation of basic GPSS/H concepts. For example, GPSS/H provides a static pool of single-server entities called Facilities. The size of this pool can be changed by a modeler, and individual members of the pool can be referenced either numerically (Facility number 1...N) or by means of symbolic names which have user- or compiler-assigned numeric values. In SLX, it is possible to have arrays of Facilities, dynamically created Facilities, pointers to Facilities, sets of Facilities, and objects which contain Facilities. While the basic concepts of what a Facility is and how it operates remain unchanged, Facilities can be used in much more general ways.

A fourth goal in the development of SLX was to eliminate, wherever possible, pitfalls encountered by users of GPSS/H. The most important pitfall we eliminated was the use of numeric references to GPSS/H entities. For example, suppose that a model of a computer system used Facilities 1...10 to model disks and Facilities 11-30 to model communications ports. If the GPSS/H model erroneously attempted to access disk number 11, which doesn't exist in the system being modeled, no run-time error would occur, because Facility number 11 *does* exist. To make matters worse, the actions intended for a disk would be performed on a communications port, since that's what Facility number 11 represents. In SLX there is no fixed pool of Facilities. Therefore, an individual Facility *cannot* be identified using only a number. SLX does, however, provide for arrays of Facilities, as described above, so arrays named "disk[1...10]" and "port[1...20]" could be used. An attempt to access disk[11] would result in a run-time error (array subscript out-of-bounds). Thus in this SLX example, a modeling/programming error maps directly into an automatically diagnosed programming error.

A fifth goal in the development of SLX was to build a platform that would be ideally suited for developing high-level, application-specific packages for use by non-simulationists. In recent years, we have had considerable commercial success providing GPSS/H and its companion animation product, Proof Animation, as unseen "engines" which are used to implement high-level packages. (See Smith and Crain (1993) and Earle and Henriksen (1993).) SLX contains a number of extensibility mechanisms, described below, which allow a package developer to adapt SLX to his or her applications. We wanted SLX to be a highly malleable tool.

A sixth goal in the development of SLX was to keep the size of SLX itself down to a minimum. Over the years, we had extended GPSS/H to a point where it contained over 140 types of statements. This was simply too big. Our experience with GPSS/H provided a powerful motivation to carefully evaluate the utility and generality of each feature under consideration for inclusion in SLX. Some features were rejected on the basis that, although they could be implemented, they could not be easily explained or learned. Ill-conceived features benefit neither the users nor the developers of a software package: users become frustrated when they can't understand complex features, and vendors become frustrated when they are overwhelmed by endless "tech support" calls .

3 THE SLX ENVIRONMENT

SLX operates in a highly interactive, window-based environment. The initial version of SLX employs the user interface that was developed for Proof Animation. We selected the Proof interface for three reasons: (1) its use provides compatibility across the products offered by our company; (2) it provides a stable environment which we control, and (3) as of this writing, the world of 32-bit graphical user interfaces lies in turmoil. The SLX environment features a fully integrated editor, compiler, and run-time supervisor/debugger. The suite of integrated tools greatly facilitates model development. For example, test compilations of models can be performed from within the editor. On "typical"

machines, "typical" models can be compiled at rates exceeding 1,000 statements per second. Compilation errors are highlighted in the source text, and tools are provided for quickly examining any errors that have been identified. The SLX debugger improves on the GPSS/H debugger by providing "watch windows" (for monitoring variables through the course of model execution) and other advanced capabilities.

4 CONCEPTS RETAINED FROM GPSS/H

The most important concepts retained from GPSS/H are its transaction-flow world-view and its basic entity classes (Schriber (1991); Banks, Carson, and Sy (1989); Henriksen and Crain (1989)). While a discussion of the transaction-flow world-view is beyond the scope of this paper, we will briefly state that this world-view has proven to be extremely flexible and powerful, and easily learned. Furthermore, departure from the world-view of GPSS/H would seriously diminish the viability of SLX as an evolutionary growth path for users of GPSS/H.

A second important concept retained from GPSS/H is its total run-time error checking and complete reproducibility of run-to-run results. Although this concept is often taken for granted, its benefits are well worth the extreme difficulties it presents to the language implementor. In a simulation, events take place in complex (usually random) circumstances. Thus it is critical that run-time errors, such as referencing beyond the end of an array, be unfailingly trapped. Furthermore, it is important that all variables be initialized to a known state. In a complex simulation, one expects to have unanticipated events. (That's a major reason for simulating; it's better to have such events occur in a model than in a real system.) Determining the cause of an unexpected event can be quite difficult. Allowing undetected programming errors to muddy the waters is totally unacceptable.

The insistence on total run-time error checking imposed great implementation difficulties for some SLX features. For example, SLX includes pointer variables, which were modeled after (in restricted form) those of the C language. In C, no automatic checks are made on the validity of pointer variables. Two of the most frequent errors made in C programs are (1) using pointers which have been assigned a NULL (zero) value, and (2) using pointers which towhich *no* values have been assigned. Such errors can lead to results which are unpredictable, if not disastrous. While the absence of error checking allows C programs to run very efficiently, it represents a rich source of pitfalls. Undetected invalid pointer references are totally unacceptable in a simulation. Hence, SLX's pointers are completely validated.

A third major concept retained from GPSS/H, and improved upon by SLX, is highly interactive debugging and execution. Many GPSS/H debugging capabilities have been retained, e.g., the ability to step through source code one or more lines at a time, the ability to set breakpoints, and the ability to interactively and selectively display critical model data. Integrating the SLX editor, compiler, and run-time supervisor have enabled us to improve upon the debugging capabilities of GPSS/H.

5 CONCEPTS MODELED AFTER OTHER LANGUAGES

A great deal of the syntax of SLX is modeled after the C language. For example, the assignment, if, else, for, while, do, switch, continue, and break statements of SLX are modeled *directly* after C, with a only a handful of restrictions imposed. Similarly, the syntax for variable names, array references, pointer variables and references, and object (struct) definitions is modeled after C. We had three reasons for choosing C as a model: (1) we felt it was better to base SLX on an existing language, rather than developing a totally new language no one knew; (2) we wanted to use a language with a high power-to-size ratio (a hallmark of C) ; and (3) we had a great deal of experience using C, and we appreciated both its strengths and its weaknesses.

While C was a good model for syntax, some of the semantics of C were deemed inappropriate for simulation. C was designed to provide systems programmers easy access to low-level, hardware-based constructs. For example, in C, pointer variables can point to atomic data elements, such as integers and characters, located virtually anywhere in machine memory. In SLX, the requirement for total run-time validation of pointers led us to disallow the use of pointers to access arbitrary atomic data types. In SLX, pointers can only point to *objects*, and, as discussed in Section 4, each such reference is validated.

Some features of C were deemed inferior to those of GPSS/H. For example, in C, character strings are treated as arrays of characters (as is the case in many languages), while GPSS/H contains string data types and true string operations, such as string assignment, concatenation, comparison, and substring extraction. Therefore, SLX *excludes* C-style character variables and *includes* an extended version of the GPSS/H string capabilities.

SLX includes "sets" and a number of set operations similar to those of Simscript (Russell (1983)). The implementation of sets in SLX differs from that of Simscript in one extremely important respect. In Simscript, the user has direct access to all the data defined by the compiler to implement any given set. Thus, users can directly manipulate the underlying data structures which implement a set. It is therefore entirely possible, through user error, to destroy the integrity of a set. In Simscript, the simulation event list is implemented as a set. Destroying the integrity of the event list can wreak absolute havoc. In SLX, basic set operations are implemented in the SLX kernel (level zero), so it is impossible for a user to damage a set.

Let us illustrate the utility of several of the concepts modeled after other languages. In GPSS/H, dynamic objects which flow through a system are modeled using Transactions. For example, in a model of a highway system, Transactions could be used to represent cars, trucks, buses, and other vehicles. In GPSS/H, all user-defined properties of Transactions are numeric values, called Parameters. All Transactions in a model have the same general format and differ only in their numbers and kinds of Parameters. (1-byte, 2-byte, and 4-byte integer, and 8-byte floating point Parameters are provided.) While this format is quite flexible, it does not allow for non-numeric properties. In SLX, a user must define each Transaction type as an SLX object. In SLX, objects can have a wide range of properties, including among others, character string values, which are very useful for naming entities. The additional step of having to define each Transaction type in SLX forces a discipline on the user. In GPSS/H, this extra step is not required; i.e., Transactions and Parameters can simply be used without being explicitly defined. While this may be easier in the short term, our experience has such undisciplined use to be a rich source of pitfalls.

In many models, it is useful for Transactions to be able to examine the properties of other Transactions. For example, in a model of a highway system, it might be necessary for a Transaction representing a car to be able to access information about other, nearby cars. In GPSS/H, Transactions can be placed in Groups or User Chains, and several statements which manipulate Groups and User Chains afford a Transaction limited access to other Transactions. The *only* access a Transaction has to other Transactions is through these statements, and the use of these statements to accomplish such access imposes some difficulties on a modeler. In SLX, Transactions are objects, and objects can be accessed by means of pointer variables. If a Transaction has access to a pointer variable which points to another Transaction, the first Transaction, by definition, has access to all of the properties of the second (other than those which are "private.")

If we were to extend GPSS/H to improve the ability of a Transaction to access another Transaction's Parameters, we would have to add new statements or SNAs (GPSS/H built-in functions) to the language. This would add to the size and complexity of a language which is already quite large, and the new features would be unique to GPSS/H. The SLX approach uses more general concepts, and these concepts are available in other programming languages. Thus the value of prior experience with other languages is likelier to be much greater with SLX than with GPSS/H.

6 WHY NOT OOP?

Although there are many languages which claim to be object-oriented, the most popular by far is C++. In the discussion which follows, we will use C++ as a representative of these languages. Our comments are confined to programming technology and do not neccessarily fully apply to the larger subject of object-oriented design (OOD) methodology. (We have many more reservations about OOP than OOD.)

Although some of the ideas in SLX were modeled after Object-Oriented Programming (OOP), SLX is not by any stretch of the imagination truly "object-oriented." One might ask why we didn't use more of the OOP technology. The answer lies in our perceptions of the GPSS/H user community. In most applications, GPSS/H is used by persons whose major professional focus is neither programming nor computer science. In addition, most applications are undertaken by very small groups of people. Most often, simulation projects are carried out by single individuals (the so-called "Lone Ranger" paradigm). In some cases, these individuals are doing their first simulation, so ease-of-learning is extremely important. Consideration of these factors led us to reject the OOP approach (as embodied by languages such as C++) for the following reasons:

Languages such as C++ are extremely complex and arcane. We felt that the benefits to be gained from OOP were outweighed by an unacceptable increase in complexity and concomitant decrease in ease-of-learning. The greatest advocates of OOP, computer scientists and very large organizations, have little in common with our users.

Languages such as C++ provide direct access to primitives such as pointers, for which no straightforward support is available for run-time validation. One could argue that simulation primitives should be encapsulated in C++ classes which do provide appropriate protections; however, if users have access to lower-level primitives, sooner or later they will exploit this access and place the integrity of the simulation run-time environment at risk.

One of the motivations behind OOP is to provide a better way to tackle very large projects. Given that the typical simulation project is done by a single individual, a big-project mentality may represent overkill. Despite its immense popularity, the OOP approach remains to be demonstrated as the great panacea of the 1990s.

Our final reason for not developing an extremely OOP-like language was that reinventing the wheel of OOP made no sense. Any attempt on our part to do so would have to compete with the products of well-established, large purveyors of OOP technology.

7 EXTENSIBILITY MECHANISMS

Extensibility mechanisms provide the means by which higher level SLX capabilities are constructed from lower level capabilities. The objective for using extensibility mechanisms is to achieve more abstract representations of data and operations on data. For example, in a model of a telecommunications system, it might be very convenient to have SEND and RECEIVE verbs. These verbs would make it easier to describe message flow at a

high level. The implementation details which define the SEND and RECEIVE verbs would be written at a lower level and (ideally) hidden from their high-level use. (The principle of information-hiding is a key component of OOP. We agree with the appropriate use of this principle. The difference between OOP disciples and us is that we don't make a big deal out of it.)

Virtually all programming languages contain procedural abstraction — the ability to define many layers of procedures. (In some languages, procedures are called subroutines or functions.) Procedures are inferior to other abstraction mechanisms in several ways. First, the readability of procedure calls is often inferior to that of in-line operators. For example "a = b + c" is more readable than "a = add(b, c)" and much more readable than "assign(a, add(b, c))". Second, in most languages, procedures have an invariant sequence of arguments, making it difficult to apply certain operations to arbitrary collections of arguments. For example, in C, the min function takes the minimum value of two integers. Thus, "min(i, j)" is the minimum of i and j. If one wants to take the minimum if three variables, i, j, and k, one must write "min(i, min(j, k))". In ANSI-standard C, it is possible to write procedures with a variable number of arguments (the "varargs" feature.) In our experience, procedures with a variable number of arguments have proven to be a rich source of pitfalls.

SLX contains a *number* of mechanisms for procedural abstraction. A user can define new *operators* which use an infix notation. For example, a SWAP operator could be defined to swap the values of two variables. "x SWAP y;" could replace the familiar "temp = x; x = y; y = temp;" 3-statement sequence. A user can also define prefix operators or even statement operators. For example, a new statement called SWAP could be defined, so that "SWAP x,y;" would be possible. Finally, SLX operators can be defined to operate on lists of arguments. These operators look very much like in-line function calls or C source macros, but offer a number of convenient extensions. For example a list operator can be defined to operate on a list with an unspecified number of items. Thus "min(i, j, k)" is possible in SLX. The definition of the "min" operator would contain a *compile-time* loop, in which the arguments of the min function would be examined in succession. Allowing a variable number of operands in list operators makes up, at least in part, for disallowing a variable number of arguments for procedures. The key difference between the two approaches is that the error-checking resources of the SLX compiler allow *compile-time* validation of variable-length argument lists, and the absence of variable-length procedure argument lists greatly stabilizes procedure calling mechanisms.

The second form of extensibility offered by SLX is *data abstraction*. In SLX, the user can define new data types (in terms of old ones) and new data structures. The data abstraction features of SLX are much closer to C than to C++. In OOP, the distinction between procedural abstraction and data abstraction is intentionally blurred. The idea is that the user of a feature needn't know how a feature is implemented; what is needed is a clear description of the interface for using the feature. The interface specification is a "contract" between the user and the developer of the feature. In SLX, in part because the developer and the user are often the same individual, a traditional (pre-OOP) distinction is made between data and procedural abstraction. This decision on our part has many consequences. As one small example, consider the notation used for invoking a parameterless procedure. In SLX, an empty parameter list must be used, e.g. "myproc( )". OOP advocates would argue against the required use of parentheses; they would believe that whether "myproc" is a stored variable or a computational procedure is immaterial to its user. SLX favors a WYSIWYG (what you see is what you get) approach.

8 CAN DO, CAN'T DO, MUST DO, SHOULD DO, SHOULDN'T DO

The value of any software tool is measured by what one can and can't do with it, what one must do to use it, and what one should or should not do with it. Evaluation of these measures is highly subjective. Consider, for example, comparing the utility of a language like GPSS/H to the utility of a high-level, application-specific software package. (Smith and Crain (1993) provide some interesting perspectives on this issue.) If a given application can be handled in its entirety by an application-specific package, and the application-specific package is available at a reasonable cost, the application-specific package is almost certainly the tool of choice. However, one must consider very carefully the possibility of encountering the "90 percent syndrome." The 90 percent syndrome occurs when a software package allows you to accomplish fairly rapidly 90% of what you have set out to do, but when you reach the 90% point, you become aware of limitations which severely impair your ability to accomplish the remaining 10% of your project. If you use a general-purpose package, accomplishing the first 90% of a may take longer than using an application-specific package, assuming the availability of such a package. However, if the general-purpose tool enables you to accomplish 100% of the task at hand, and the application-specific package does not, the general-purpose package is superior. The "overhead" of failure is infinite.

The layered architecture of SLX casts the "can do - can't do" question in a somewhat different light. With SLX, the operative rule is to use the highest levels as much as possible. For example, if an SLX-based, application specific package exists for the problem you are dealing with, you should by all means use the package. However, if you find that the package has some shortcomings when you reach the 90% point of project completion, you have recourse to lower levels of

SLX. Ultimately, you have access all the way down to the SLX kernel. At each level, SLX provides error detection which guarantees the integrity of fundamental data structures.

There are other simulation software systems which offer a layered architecture. For example, in some languages, a user can write C or Fortran subroutines to augment the capabilities of the language. Unfortunately, the use of C or Fortran entails plunging into a world plagued by an overabundance of complexity and a lack of protections against damaging fundamental data structures. The SLX kernel provides a totally secure, powerful, low-level implementation language. With rare exception, there is no need to go below the level of the SLX kernel. Furthermore, the extensibility mechanisms of SLX facilitate moving from layer to layer.

A language can pass the "can do" test but fail the "must do" test. The "must do" test identifies capabilities not provided by the language, which a user must program for himself or herself. Simula/67 (Birtwistle, Dahl, Myhrhaug, and Nygaard (1973)) provides a classic example. Simula was (and is) one of the great simulation languages, and it holds the distinction of being the grandfather of all OOP languages. Unfortunately, Simula has achieved limited success in the United States. This is due in part to the fact that Simula is an Algol-based language, and Algol has achieved limited success in the U.S. However, another significant reason for its limited success is attributable to the "must do" criterion. Proponents of Simula emphasize all the things one *can* do with the language. Detractors point out all the things one *must* do in order to use the language. For example, the original language provided no capabilities for automatically generating simulation reports. Reports had to be built by the end user, "from scratch," albeit in Simula. By comparison, even the earliest versions of GPSS had automatically-generated reports. In SLX, we have tried to provide a reasonable collection of modeling tools, so that what you can do with SLX is unlikely to be overshadowed by what you must do.

Finally, one must consider the "should do - shouldn't do" comparison. Although SLX has a very powerful, general-purpose kernel, clearly it is heavily influenced by the fact that its purpose is to be used for building discrete event simulation models. SLX is very good for applications which fall within its scope. The scope of applicability of SLX is considerably broader than that of GPSS/H.

9 SUMMARY

SLX is a new, layered system for building discrete event simulation models. Its middle layers are heavily influenced by GPSS/H, and its lower layers are heavily influenced by the C language. The most important features of SLX are its extensibility mechanisms, which facilitate the construction of higher layers from components contained in lower layers. Users of SLX are encouraged to use the highest layers suitable to their application. However, if they reach a point where an upper layer lacks a needed capability, they have recourse, through use of components contained in lower layers. The foundation of SLX is a C-like, powerful kernel language which enforces strict compile-time and run-time validation

REFERENCES

Banks, J., Carson, J. S. II, and Sy, J. N., 1989. *Getting Started with GPSS/H.* Annandale, Virginia: Wolverine Software Corporation

Birtwistle, G., Dahl, O., Myrhaug, B., and Nygaard, K., 1973. *Simula Begin.* New York: Petrocelli/ Charter

Henriksen, J. O., and Crain, R. C., 1989. *GPSS/H Reference Manual.* Third Edition. Annandale, Virginia: Wolverine Software Corporation

Henriksen, J.O. and N.J. Earle. 1993. Proof Animation: Better Animation for Your Simulation. In *Proceedings of the 1993 Winter Simulation Conference*, eds. Evans, G., Mollaghasimi, M. Russell, E., and Biles, W. Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

Russell, E. C., 1983. *Building Simulation Models with Simscript II.5.* Los Angeles: CACI, Inc.-Federal.

Schriber, T. J., 1991. *An Introduction to Simulation Using GPSS/H.* New York: John Wiley & Sons.

Smith, D. S. and Crain, R. C., 1993. Industrial Strength Simulation Using GPSS/H. In *Proceedings of the 1993 Winter Simulation Conference*, eds. Evans, G., Mollaghasimi, M. Russell, E., and Biles, W. Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

AUTHOR BIOGRAPHY

James O. Henriksen is the president of Wolverine Software Corporation, located in Annandale, Virginia (a suburb of Washington, D.C.) He is a frequent contributor to the literature on simulation and has presented many papers at the Winter Simulation Conference. He served as the Business Chair of the 1981 Winter Simulation Conference and as the General Chair of the 1986 Winter Simulation Conference, and he has served on the Board of Directors of the conference as the ACM representative. He was a keynote speaker at the 1992 twenty-fifth anniversary Winter Simulation Conference.