*Proceedings of the 1996 Winter Simulation Conference*
*ed. J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain*

# THE POWER AND PERFORMANCE OF PROOF ANIMATION

James O. Henriksen

Wolverine Software Corporation
7617 Little River Turnpike
Annandale, Virginia 22003, U.S.A.

## ABSTRACT

The Proof Animation™ 3.x product family provides tools for animating a vast array of applications. There are versions of Proof Animation to meet any need, from small and mid-sized projects to large scale applications. The Proof product family runs on readily available, inexpensive PC hardware. Proof is ASCII-file-driven and features a general-purpose design. Its vector-based geometry provides a large animation canvas with the ability to zoom in or out while maintaining crisp, clear images. Proof's features include post-processing for maximum performance, built-in drawing tools and CAD import/export for ease of creating animation layouts, dynamic bar graphs and plots for displaying statistics, multiple-window display, and a unique presentation-making capability. Proof's open architecture makes it ideally suited for serving as an animation engine for models written in a wide variety of simulation and programming languages. Proof's superior power and performance assures smooth, realistic motion for animations regardless of their size, complexity, or application.

## 1 INTRODUCTION

Animation software has been around for many years; the concept is not new. Animation is often considered a requirement to a simulation study because it is a meaningful, proven way to show results to an audience with varied backgrounds. As more animation products enter the market, a user must make sure that the product purchased can meet the task at hand, as well as any future projects. Some animation software is limited to a single type of application. Moreover, some of these animation packages may only be useful for a specific type of problem *within* that application. In almost any application, there are many areas that benefit from simulation and animation. These can range from capacity analysis to scheduling, staffing, and more. Some application-specific animation software may not be able to handle the full range of problems even within a *specific* application. Buying single-application animation software can be costly, especially if a user does projects within a wide range of applications.

Proof Animation is a powerful general purpose animation tool. It is not tied to a specific area or application. Proof Animation can be used to animate the full range of applications from areas such as Business Process Reengineering to the classic applications such as healthcare, manufacturing, and traffic. The size of the system to be animated is not an issue when using Proof Animation. There are versions of Proof Animation that can be used to animate small to mid-size systems as well as a version to handle the largest systems.

A user may want to add animation to an already existing model. Since time is rarely unconstrained, he or she does not want to have to change any existing and valid model logic to make the animation look realistic. Some animation tools are so limited that it is difficult or even impossible to produce an animation that can mimic the motion of the real-world system. Wolverine Software Corporation's goal has always been to design animation software that is easy to use and able to produce realistic animations that help sell the results of simulation projects.

Another issue in animating existing models is interoperability. Can the animation software work with the existing simulation software? Proof Animation can be paired with most simulation and programming software. The user can continue to use the modeling tool with which the existing models were written.

Proof Animation 3.1 has an advanced feature set unparalleled in its ability to meet the challenges of animating today's complex systems while keeping ease of use a top priority. In the following sections we describe the products, discuss their design, give an overview of how they are used, and describe the advanced features.

## 2 THE PROOF ANIMATION FAMILY

All of the Proof Animation products run on readily available and inexpensive PC hardware. All run as 32-bit applications that require only a 386 or better CPU, a math coprocessor, and at least a VGA-compatible video card. Because this basic configuration is very widely available, the portability of animations developed with Proof is maximized. It is very likely that this hardware configuration will be available at any site.

The Proof Animation family runs under DOS and can also be launched as a full-screen application under Windows 3.x, Windows 95, Windows NT, or OS/2. A set of Windows icons are supplied with each of the Proof Animation products to provide single-click launching. The following products comprise the Proof Animation family:

- PROOF ANIMATION
  Proof Animation is the basic animator. Memory size is fixed and limited. Includes built-in CAD import/export feature. Good for small to mid-sized animations.

- PROOF PROFESSIONAL
  Proof Professional exploits all available extended memory for animating large systems. Includes built-in CAD import/export feature.

- RUN-TIME PROOF PROFESSIONAL
  Run-time Proof Professional runs developed animations or presentations, but has no animation *development* capabilities. It provides a low cost way to run different scenarios with a fixed layout file prepared using Proof Professional or Proof Animation.

- STUDENT PROOF ANIMATION
  The student version of Proof Animation is included with the *Using Proof Animation* text. Size and playing time limitations are imposed; otherwise it is identical to Proof Animation.

- PROOF ANIMATION DEMO MAKER
  Demo versions of animations can be prepared under a licensed copy of Proof Animation or Proof Professional containing the Demo-Maker add-on. Copies of the executable demo files can be reproduced and distributed free of charge and viewed by anyone. No licensed copy of Proof is needed to *view* a demo prepared with the Demo Maker.

## 3 MAXIMUM PERFORMANCE DESIGN

### 3.1 The General-Purpose Approach

One of the ways in which the performance of Proof Animation is maximized is its general-purpose design. A general-purpose animation product can be used without being tied to a specific simulation or programming language. While built to work easily with Wolverine's GPSS/H simulation software, Proof Animation also provides affordable and powerful animation software to users who develop models in other simulation and programming languages.

Most animation software from other vendors is directly coupled to their simulation software. In other words, one cannot use *their* animation software without also using *their* simulation software. In some cases, the simulation and animation software are sold only as a pair, so both must be purchased regardless of the needs of the user. The suggested advantage of the coupled approach is that because the animator has direct access to the simulation events, development of the animation is supposedly simplified. However, the real advantage is felt by the vendors. They sell more software since their users do not have an option to pick and choose based on their needs. Moreover, a user of the coupled software has little or no control over what information is passed to the animation; therefore, he or she may actually have to alter the modeling approach used in the simulation to achieve the desired appearance and level of detail for the animation.

Another disadvantage of the tightly coupled simulation/animation package is cost. Sole sources tend to be expensive. Vendors of these tightly coupled packages often claim that their approach is the *only* way to add animation to a simulation. Proof Animation has proved that wrong. The number of success stories using Proof Animation with other software continues to grow. Furthermore, a benefit of the mix-and-match strategy for software purchases is that the selection can be based on optimal functionality and price.

#### 3.1.1 ASCII Input Files

Proof Animation can be used with other software because of two major design features. One is that Proof Animation is driven by ASCII files. Therefore, any software capable of writing ASCII text files can be used with Proof Animation.

Proof Animation requires two ASCII input files to run an animation: the layout file and the trace file. The layout file describes the geometric details of the background over which objects move, provides

geometric definitions and properties for such objects, and defines logical paths along which the objects move.

Ordinarily, layout files are produced at least in part by using Proof Animation's drawing tools; however, the layout file command set specifications are published so programs can easily be written to generate layout files. For example, some users have written front ends for their simulation models that allow different system design parameters to be specified for each run. Based on these parameters, different geometric configurations are written and incorporated into a layout file. The new layout appears on screen when Proof Animation is invoked.

The trace file contains a time-ordered sequence of commands such as CREATE, DESTROY, PLACE, PLOT, MOVE, SET SPEED, SET COLOR and many more. This file provides Proof Animation with information on when, where, and what to create, destroy, place, plot, etc. Trace files are free-format, and the commands are easily learned and used. They provide exactly the kind of flexibility necessary to easily be integrated with the simulation model logic. Any language that can produce formatted ASCII output can write a trace file.

### 3.1.2 Post-Processor Animation

The second of the two design features that make Proof Animation compatible with other software is that it is a post-processing animation package. Post-processing means that it runs *after* the simulation has executed. Both the layout and trace files must exist before invoking Proof Animation. They cannot be written and read concurrently.

Two great advantages result from the post-processing approach. First, PC hardware is not shared between the simulation and the animation. This leaves the entire CPU for running the animation. Second, it provides the abilities to jump back and forth in time during the animation playback, to speed up or slow down the viewing speed, or show all or a specific portion of an animation. These features make it easy to investigate unusual system behavior or highlight points of interest.

## 3.2 Vector-Based Geometry

In the Proof Animation product family all layout file information is based on vector geometry. Vector-based descriptions are automatically mapped into the screen's pixels to build the image. One of the advantages of this approach is that layouts can be much larger than a single screen. With the ability to zoom in or out and pan, larger layouts are easily navigated to show the overall layout or zoomed in to whatever level of detail is necessary. Vector-based geometry also provides the ability to have moving objects realistically *rotate* around corners instead of the sliding effect to which other animation packages are limited.

Another advantage of vector-based geometry is that many CAD packages are capable of producing standard vector-based .DXF files. In many cases, a CAD drawing already exists for the system to be animated. If that is the case, the effort of redrawing an entire layout can be avoided. Proof Animation's built-in CAD Import/Export feature provides the capability to convert industry-standard .DXF files into Proof Animation layout files, *and vice versa*. Credibility of the study is enhanced when viewers see a familiar CAD drawing of the system integrated into the animation. These advantages maximize the power of the animation by giving a user total flexibility on the detail and complexity of the drawing.

Instead of vector geometry, animation packages may use a pixel-oriented approach for drawing. With the ability to manipulate individual pixels, one can produce detailed images. However, this level of detail is time consuming to draw and can often be lost because of the scale at which the animations are viewed. Some other disadvantages of pixel-orientation are: (1) pixel-oriented images cannot be rotated; (2) layouts are often confined to single-screen images. Some animators offer multi-screen operation; however, the individual screens are disjoint and independent, unlike Proof Animation's single, continuous canvas; (3) zooming in on pixel images magnifies the jagged edges inherent in all such images. When a zoom in is performed in Proof Animation, the vector-based image maintains its crisp and clear appearance. Lines continue to look like lines. If a zoom in is performed on a pixel-based animation layout, the effect of the jagged edged image makes a line look more like a stairway.

## 3.3 Smooth Motion

The maximum performance design of Proof Animation achieves very smooth motion. Proof Animation maintains this smooth motion by updating or refreshing the screen 60-70 times per second. Other software can often sustain refresh rates of only 5-10 updates per second. The ultimate purpose of an animation is to achieve a realistic depiction of the system being studied, allowing the audience to gain confidence in the results of the simulation study. Objects that move smoothly across the screen are more realistic than those that jump across the screen.

### 3.4 Color and Resolution Options

Beginning with Release 3.0, all versions of Proof, including the student version, now provide for operation in 256-color mode in a variety of screen resolutions. Prior to Release 3.0, "standard" versions of Proof operated in 16-color mode at 640x350 resolution. 256-color mode and higher screen resolutions were provided as no-cost options only to users of Proof Professional. Support was limited to about a half dozen video chip families.

The "normal" operating mode for Proof is 256-color mode, and the "normal" resolution is 640x480. Video hardware which supports this mode and resolution is very commonplace. Proof can also run at 800x600 and 1024x768 resolutions, provided enough video memory is available. These resolutions are now even available on some *laptop* machines. For older hardware with less video memory, Proof can run using 640x400 resolution. For very old hardware, Proof can still be run in its original 640x350, 16-color mode.

256-color animations are *much* more attractive than 16-color animations. The expanded color palette contains 24 foreground colors and 8 background colors. The background colors consist of 7 layout colors and 1 backdrop color. The background colors do not interfere with the foreground colors and therefore give a user much more flexibility when drawing the static background portion of the layout. With more colors from which to choose, the background can be drawn with more detail without sacrificing the color integrity. In addition, in 256-color mode, objects can be multi-colored. (In 16-color mode, objects must be mono-chrome.)

## 4 NAVIGATING PROOF'S MODES AND MENUS

Proof Animation is organized into seven menu-driven *modes*. Each mode is a collection of closely related functions. Switching among these functions is very easy. Usually, a single mouse click is all that is required. Switching among *modes* is also easily done, but it implies major changes of context. For example, running an animation and drawing a layout are vastly different activities. Each mode has one or two main menu bars at the top of the screen. Clicking on main menu items invokes the options for the lower level tools. The seven modes are summarized as follows:

- RUN MODE
  This is the mode in which animations are viewed. It provides menu tools for starting and stopping an animation, changing views, controlling viewing speed, jumping ahead and back in time, and more.

- DEBUG MODE
  This mode provides tools for stepping through an animation by individual events or time commands and examining the resulting movement. Information pertaining to an individual object can be obtained by clicking on the object with the mouse.

- DRAW MODE
  This mode contains the drawing tools used for creating the layout background. Tools are provided for drawing static elements such as lines, arcs, text, fills, etc. Dynamic elements such as messages, bars, plots, and layout objects are defined in Draw Mode.

- CLASS MODE
  This mode is used for defining object classes. A class serves as a template for creating both the dynamic objects that move around a layout and layout objects that generally remain stationary. The template determines an object's size and shape and other initial properties such as default speed and color. An animation will usually contain multiple object classes. For example, an animation of a hospital might contain classes that represent doctors, nurses, equipment, and patients.

- PATH MODE
  This mode is used for defining fixed paths. A path is a perfect mechanism for describing for guided, directional movement such as conveyors. The geometry or route of a path is easily defined by clicking on existing lines and arcs of a layout. Tools are also provided for defining path speeds, circularity, and accumulation status. Accumulating paths provide automatic queuing for objects that pile up at the end of the path.

- PRESENTATION MODE
  This mode is used for running scripted presentations. Scripts can include static bitmap slides and snippets of animation, separated by special effect segues such as screen fades and dissolves.

- SETUP MODE
  This mode is used for examining and altering infrequently changed configuration data. For

example, the color palette can be customized or the mouse speed changed in this mode.

## 5 CREATING ANIMATIONS AND PRESENTATIONS

### 5.1 Drawing the Layout

The first step in developing an animation is to draw a layout. If a .DXF formatted CAD drawing of the system is available, a user can begin by importing the drawing into a Proof Animation layout file. This is done using the built-in CAD import/export utility. Once imported, the drawing can be examined by layer or by line style. Specific layers and line styles can be deleted from the drawing as desired. When you save the resulting drawing, it is saved as a Proof Animation layout file. The original .DXF file remains intact.

If a user does not have a CAD drawing or prefers to draw using a computer, the drawing tools provided in Draw Mode are easy to use. Although it is mouse-oriented, Draw Mode also allows keyboard input, so if a user needs to draw a line of a specific length at exactly a certain angle, he or she can enter these specifications *numerically*, and the geometry will appear on the screen. To help in drawing scaled, accurate layouts, a visible grid is turned on automatically when Draw Mode is entered. For additional aid in drawing, Proof Animation has the Snap-to-Grid option. This option is also *on* as the default setting. Snap-to-Grid limits the drawing of layout elements from grid point to grid point, thus eliminating the chance of small gaps between the endpoints of *seemingly* connected lines. Other snap options which help draw accurate layouts are Snap-to-Endpoint which *magnetically* attracts the mouse cursor to the ends of lines and arcs, and Snap-to-Tangent which quickly finds points of tangency between lines and arcs. All of these options can be turned on or off by the user during the drawing session.

### 5.2 Defining Object Classes

Once the background of the animation is drawn, the second step in developing an animation is usually to define one or more object classes. This is done in Class Mode. Objects and object classes are among the most important constructs in Proof Animation. A class provides the geometric description of the individual objects that move throughout the animation. The class definition also includes the initial properties such as physical clearances, color, and speed of the individual objects. Each animation will usually have a collection of object classes.

It is helpful to think of an object class as the template from which the individual objects are made. An individual object is based on the single geometric description of a particular object class. There can be an arbitrary number of *objects*, such as widgets, in the system at once, but there need be only *one* widget object class.

Motion and color-changing commands in the trace file operate on *objects*. The drawn background components, produced in Draw Mode, cannot be moved or changed. If dynamic changes in background elements are required, the appropriate components must first be defined as object classes and can then be created and positioned directly in Draw Mode. Objects that are created and placed in the layout while the user is drawing the background are called *layout objects*. Layout objects enable a user to scale and position the objects into the layout while having the background components visible as reference points. While the animation is running, layout objects can be manipulated using trace file commands. For example, if an idle machine is shown as green and a busy machine as red, the machine must first be defined as an object class. Objects from that class can be created and placed as part of the layout file, and their color can be changed while the animation is executing.

### 5.3 Defining Paths

Proof Animation provides two kinds of motion: absolute and guided. Absolute motion, specified by the MOVE trace file command, causes an object to be moved between two points. Guided motion always occurs along a fixed route, called a path. If objects will follow guided motion, such as travel on conveyors or along guide wires, the next step in the animation development is to use Path Mode to define one or more paths.

Paths are comprised of lines and arcs that represent the route that the objects will follow. This underlying geometry must first be drawn using Draw Mode or imported from CAD. The logical path segments are then defined *on top* of the existing lines and arcs. A single line or arc can be part of one or more paths. Once defined, paths are saved as part of the layout file.

Using paths is very simple because Proof Animation does all the work. The most commonly used trace file path command is PLACE *object*ID ON *path*. Once an object is placed on a path, it will follow that path until it visually comes to rest at the end of the path or until it is PLACEd elsewhere or DESTROYed. All objects traveling on the same path can be stopped simultaneously and resume movement at a later time.

Paths provide outstanding animation power in response to a single trace file command.

*Accumulating* paths provide even greater power for animating paths on which queuing can take place. On accumulating paths, Proof Animation reflects physical reality by *visually* queuing objects when bottlenecks occur. This often makes a simulation model of the system much simpler to construct, because such queuing need not always be explicitly represented in the model code. Most systems contain some accumulation. This property can be used to represent certain types of conveyors, cars at a traffic signal, bank lines, and more. Paths play an especially important part in transportation, product flow, and material handling animations.

### 5.4 Writing the Trace File

The next step in the animation development is producing the animation trace file. Trace files consist of very readable ASCII commands. Trace files are time ordered. That means the specific animation events take place between TIME commands. Consider the following portion of a trace file:

```
TIME 34.6
CREATE PLANE 1
PLACE 1 ON RUNWAY3
SET 1 SPEED 75
TIME 52.8
```

It is very easy to visualize the results of these commands. At time 34.6, an object with an id number of 1 is created with geometry and properties inherited from class PLANE. This object will appear on screen at the beginning of a path named RUNWAY3 and begin moving along the path. The speed at which object 1 will move is set to 75 units of distance per unit of simulated time. These units are user-determined, e.g. feet and seconds. Proof Animation will continue reading trace file commands until it reads the TIME 52.8 command, signaling the end of the events that are to begin at time 34.6. It is very easy to produce simple trace files with any ASCII editor.

For most applications, it is impractical to create trace files by hand. Using a simulation model or program to generate the trace file is usually the *only* viable approach. In order to produce a trace file, output statements are inserted into the simulation model to write the appropriately formatted commands. The Proof Animation trace file command set has been designed to be easily generated. Any language with the ability to write a formatted ASCII file is capable of producing a trace file.

### 5.5 Building a Presentation

As an optional final step, a professional looking presentation can be built using Proof Animation. Presentation Mode lets users display scripted sequences consisting of bit-mapped screen images or slides, full animations, and/or segments selected from full animations. These presentation elements can be linked together using fades, dissolves, and other special effects, to produce a polished presentation. This is done by writing a simple ASCII presentation script file. Complete presentations can be viewed without ever exiting Proof Animation. This eliminates the awkwardness of switching back and forth between the computer and other display media during a presentation.

Slides can be created directly in Proof Animation or any software package capable of exporting industry-standard .PCX image files. There are many such packages available, and virtually all of them can produce very high-quality charts, graphs, and slides. Proof Animation can both read and write these .PCX screen images. It is very straightforward to save Proof Animation screen images as .PCX files and incorporate them into presentations as slides.

Presentations can be developed so that slides and animations appear on the screen for a defined amount of time. The viewer does not have to interact with the computer for the presentation to continue. Presentations can also be developed to continue once a key or mouse button is pressed, giving the viewer or presenter ample time to comment on what is currently on the screen.

When developing the presentation, a user can choose to highlight areas of interest within the animation by using different views or sound to draw the viewer's attention to particular aspects of the animation. Presentations can incorporate selectable menus defined by the presentation developer. These menus can be set up by topic, giving the viewer or presenter complete control and flexibility of what to show.

## 6 THE PROOF PROFESSIONAL ADVANTAGE

Proof Professional offers obvious advantages because it is limited only by the total memory available on a computer. No artificial memory limits are imposed; therefore, large-scale animations can be run on the PC.

Although Proof Professional runs on 386-based machines with a math coprocessor and VGA display, Pentium-bases PCs are the best platforms for running large-scale animations.

## 7 THE ADVANCED FEATURE SET

Advanced features make Proof Animation unparalleled in its capabilities. Keeping the user in mind, many of these features were added because of direct requests from our customers. This feature set is described below.

### 7.1 Multiple-Window Display

The animation screen can be divided into separate windows. Within each window, the view can be independently manipulated using zooms, pans and rotations to include a portion or all of the animation canvas.

With this feature users can maintain a window that keeps updating statistics in constant view while panning and zooming to different areas of the layout. For example, a user may choose to create a layout object that represents a clock to display the current system time and place it in a separate window that is always in view.

### 7.2 Displaying Dynamic Data

Along with bars, dynamic plots can be incorporated into a layout as a way of displaying statistics. A plot can be used to show any type of data displayed on X-Y axes. Plots, like bars, are defined and placed in the layout using Draw Mode. The data are displayed in the form of line segments that can be added, erased or changed via the new PLOT trace file command. A single plot can have many different types of data plotted simultaneously with each curve plotted in a different color. Plots offer a unique way of displaying changes over time. A viewer can look at a plot and see the differences in the value of the statistic as the animation progresses.

Textual messages can be incorporated into object classes. Each object created from that class will carry its own messages. The text displayed in the messages can be changed using the WRITE trace file command. Messages can be used to visually differentiate objects that are otherwise identical in appearance.

### 7.3 Features Affecting Object Movement

When an object is placed on a path, that object is positioned on the path at the object's hotpoint. The hotpoint of every object class is the point (0,0) as viewed in Class Mode. Placement of the hotpoint within the class is determined by how the geometry is drawn around that point. For example, the class can be drawn so that (0,0) is in the center, on an edge, or even outside of the geometry. Unlike compact objects, long, thin objects rounding corners of paths can exhibit a *fishtailing* effect, especially when the hotpoint is near the leading edge of the object

Proof Animation provides the ability to supply a second point of attachment to object classes. This point is called the rear guide point or RGP. The optional RGP is defined as a property of a class in Class Mode. It is input as a negative number which serves as the RGP's displacement from the hotpoint. Both the RGP and hotpoint will remain connected to a given path as long as such connections are physically feasible

Another feature that can affect the movement of objects is the ability to specify a negative speed or travel time for objects traveling on non-accumulating paths. Specifying a negative speed causes the object to travel in the reverse direction on a given path. This gives users the ability to have an object back into an area and pull out or vice versa. This is especially useful in animations that contain traffic of any type.

The PLACE...ON commands for objects entering paths has a new option called SQUEEZE. When SQUEEZE is used with a PLACE...ON, all objects on the path *behind* the point at which the new object will enter are pushed back, allowing sufficient room for the new object. Objects can be SQUEEZEd at the end or at any offset of a path as well as before or after a particular object already residing on the path. This feature is ideal for animating conveyors that are loaded manually.

The addition of the ATTACH trace file command lets the user connect objects to one another. All the objects then follow the movement of the leader object. There is a single level of attachment. For example, Object 2 can be attached to Object 1, but Object 3 cannot also be attached to Object 1 while Object 2 is still attached. To create a string of objects, a user can simply attach 2 to 1 and then attach 3 to 2, and so on. The movement of all of the objects is controlled by the first object in the string, in this case, Object 1. The trace file commands need only manipulate the leader object, and all the other attached objects will follow. Speed is also determined by the leader of the chain of objects. This new feature makes it simple to animate multi-car trains or vehicles in tow. The DETACH command is used to break the chain of objects at any point of attachment.

### 7.4 Features That Enhance The Trace File

A JUMP option has been added to the TIME command. If JUMP is used, the trace file is automatically fast-forwarded until a TIME command is encountered with a value that is equal to or greater than the value specified after the JUMP keyword. For example, a TIME JUMP 100 command issued in the trace file will fast forward

the animation until it executed a TIME 100 or higher TIME command. The animation then begins to run at the specified viewing speed, processing trace file commands normally. This is a way to skip large warm-up periods in an animation without needing to stop and enter a fast-forward time via the Run Mode menu.

There is an option in the file menu that allows the writing of an *abridged* form of the trace file. When creating an abridged trace file, a full version of the trace file must first exist. A user determines the start and end times of the trace file that will be incorporated into the abridged version. This is especially useful in preparing animations for presentations. The abridged trace file contains the information needed to immediately update the state of the animation to the start time specified in the abridged trace file and continue running normally until the user-specified end time is reached. The abridged trace file eliminates any delays that would be incurred because of the additional trace file commands that are executed during a standard fast-forward.

## 8 SUMMARY

Wolverine Software's Proof Animation has set the standard for maximum power and performance. Some of Proof Animation's many unique features include the ability to show statistics using bar graphs and plots, create presentations, built-in CAD import/export, drawing tools, smooth motion, and a unique multi-windowing display.

Proof Animation is not tied to a specific application. There are features that make it an ideal choice for the animation of systems like computer networks, health care, transportation, reengineering, manufacturing, and more while maintaining ease of use.

An animation benefits a user in every phase of the study: verification, validation, presentation of results, and the overall system design process. Proof Animation's unmatched features make it the perfect tool for each of these phases regardless of the application.

## REFERENCES

Earle, N.J. and Henriksen, J.O. 1994. Proof animation: reaching new heights in animation. *Proceedings of the 1994 Winter Simulation Conference*, eds. J. Tew, S. Manivannan, D. Sadowski, and A. Seila, 509-516 Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

Wolverine Software. 1995. *Using Proof Animation (Second Edition)*. Annandale, Virginia: Wolverine Software Corporation.

## AUTHOR BIOGRAPHY

**JAMES O. HENRIKSEN** is the President of Wolverine Software Corporation. He is a frequent contributor to the literature on simulation. Mr. Henriksen served as the Business Chairman of the 1981 Winter Simulation Conference and as the General Chairman of the 1986 Winter Simulation Conference. He has also served on the Board of Directors of the conference as the ACM/SIGSIM representative.

*Proceedings of the 1996 Winter Simulation Conference*
*ed. J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain*

# AN INTRODUCTION TO SLX™

James O. Henriksen

Wolverine Software Corporation
7617 Little River Turnpike, Suite 900
Annandale, VA 22003-2603, U.S.A.

## ABSTRACT

SLX is Wolverine Software's "next generation" simulation language. SLX builds on the strengths of Wolverine's GPSS/H (Henriksen & Crain, 1996). It provides powerful simulation capabilities in a modern, C-like language framework. SLX provides a multiplicity of layers, ranging from the SLX kernel, at the bottom, through traditional simulation languages, e.g., GPSS/H, in the middle, to application-specific language dialects and extensions at the top. This paper provides an overview of SLX for readers who are already familiar with simulation. An earlier paper (Henriksen 1995) presented key concepts of the architecture of SLX. As of this writing, SLX has been heavily used by a number of alpha testers. Insights gained from our interactions with the software testers will be presented.

## 1 INTRODUCTION

The most important characteristic of SLX is its layered architecture. The success of SLX's layered approach depends on several factors:

A. The layers are well-conceived. In developing SLX, we have had the luxury of drawing on years of experience with GPSS/H. A great deal of SLX is based on GPSS/H. In some cases, source code from GPSS/H has been directly "lifted" for use in SLX. In other cases, we have modified, simplified, or adapted GPSS/H algorithms. In a few cases, we have eliminated pitfalls and shortcomings of GPSS/H. The end result is an extremely well-designed system.

B. The layers are not too far apart. Many other languages provide multiple layers, but typically there are wide gulfs between the layers. For example, a language might provide flowchart-oriented building blocks as its primary modeling paradigm, but also provide for "dropping down" into procedural languages such as C or FORTRAN. The problem with this approach is that there are only two layers, and they are too far apart. One must become familiar with many details of the C or FORTRAN *implementation* of the simulation language to be able to add C or FORTRAN extensions. Even worse, virtually none of the error checking and other safeguards provided in the simulation language are available in C or FORTRAN. The SLX kernel language is a powerful, C-like language, so users of SLX almost never find it necessary to drop down into a lower-level, more powerful language. Furthermore, the SLX kernel language includes complete checking to prevent errors such as referencing beyond the end of an array and using invalid pointer variables. The layers above the SLX kernel exploit kernel capabilities in straightforward ways. Transitions from layer to layer are very smooth.

C. The mechanisms for moving from layer to layer are very powerful. These mechanisms are *abstraction* mechanisms. A "higher level" entity provides a more abstract description than a "lower level" entity. Lower level implementation details are hidden at the upper levels. SLX provides both data and procedural abstraction mechanisms. Like C, SLX provides the ability to define new data types, and to build objects which are aggregations of data types. The procedural abstraction mechanisms of SLX are extremely powerful. SLX provides a macro language and a statement definition capability which allows introduction of new *statements* into SLX. (The SLX-hosted implementation of GPSS/H makes heavy use of the statement definition feature.) The definitions of macros and statements can contain extensive logic, including conditional expansion, looping, optional arguments, lists of arguments, etc. In fact, such definitions are actually *compiled* by SLX, allowing use of virtually all kernel-level statements. Macros and statement definitions offer far more than simple text substitution.

In the sections which follow, selected features of the SLX kernel are presented. Following the presentation of the SLX kernel, SLX's extensibility mechanisms are illustrated. Finally, brief presentations of how SLX has been used and how SLX is taught are presented.

## 2 SLX KERNEL FEATURES

The number of primitives required to support simulation is surprisingly small. Implementing some of these primitives in a general form, however, can be very difficult. Features such as SLX's generalized *wait until* are extremely difficult to implement. Not surprisingly, this feature has rarely appeared in other simulation software. Paradoxically, some of the features which are the most difficult to implement are the most easily understood. In the remainder of Section 2, we will present some representative features, to illustrate the functionality, ease-of-of-use, and *ease-of-learning* of SLX.

### 2.1 Objects and Pointers to Objects

The very mention of the word "object" inspires a wide range of expectations and emotions, due to the widespread influence of object-oriented programming (OOP). SLX has been influenced by OOP and incorporates some OOP ideas, but *SLX is not a truly object-oriented language*. Such a statement is a rarity in this day. Many products claiming to be object-oriented are far from it. In fact, although we do not claim that SLX is object-oriented, it is probably more so than some products for which OOP architecture is claimed.

In SLX, objects are used in two ways. *Passive* objects are used for modeling entities which have no "executable" behavior. For example, a parking lot could be modeled as a passive object. GPSS/H Facilities, Queues, and Storages are implemented as passive SLX objects. *Active* objects have executable behavior patterns. Customers in a supermarket are a good example of entities that would probably be modeled as active objects. SLX active objects are roughly equivalent to GPSS/H transactions. Some entities can be modeled either as active objects or passive objects. For example, a simple server with a FIFO queue can be modeled as a passive object. Its behavior depends solely on the requests made for it by active objects. (This is the way Facilities work in GPSS/H.) For more complicated servers, an active object may be more appropriate. Consider a butcher in a model of a supermarket. In a simple queueing model, the butcher can be represented as a passive object, responding to requests for service one customer at a time. In a more realistic model, the butcher would have a more complex behavior pattern, cycling through activities of cutting meat, arranging products in refrigerators, interacting with the deli department, taking breaks, etc. Such behavior would require modeling the butcher as an active object.

SLX objects can have a number of *standard properties*. All standard properties are comprised of explicitly identified sections of executable code. The *initial* property is invoked when an object is created. The *final* property is invoked when an object is about to be destroyed. The *report* property is invoked by the *report* statement; it is used for the obvious purpose. The *actions* property specifies the behavior pattern for an active object. It is invoked by the *activate* statement (discussed in the next section). A sample object follows:

```
object customer
  {
  integer  number_of_items;

  initial
    {
    number_of_items =
    rv_uniform(stream1, 10, 20);
    }

  actions
    {
    ...
    }

  report
    {
    print (ME, number_of_items)
    "* exiting, items purchased: *\n";
    }
  };
```

Objects are created by using the *new* operator, which returns a pointer to the newly created object:

```
pointer(customer)       cust;
cust = new customer;
```

The initial property of the customer is executed before the pointer to the new customer is assigned to `cust`. The initial property can be thought of as an easier-to-use counterpart to a C++ constructor. All uses of pointers are validated to ensure that pointers always point to objects of the proper type (or contain a NULL value). Use counts are maintained for all objects. If pointer1 and pointer2 are pointer variables, an assignment statement of the form "pointer1 = pointer2" causes the use count for the object pointed to by pointer1 to be decremented and the use count for the object pointed to by pointer2 to be incremented. Storage for an

object is released if and only if its use count goes to zero.

### 2.2 Active Objects

An active object has an *actions* property. When the *activate* verb is used, a *puck* is created for the object and placed on the active puck list; i.e., the puck is placed in a ready-to-execute state. Pucks are the schedulable entities in SLX. Scheduled time delays and state-based delays, e.g., waiting for a server to become available, are puck-based operations. Thus manipulation of pucks is the basic mechanism by which a collection of objects experiences events over time. Pucks embody the means of achieving simulated parallelism.

Two kinds of parallelism can be described in SLX. Large-scale parallelism consists of interactions among active objects. For example, in a model of a freeway toll booth system, active car, truck, and bus objects would all interact. Small-scale or *local* parallelism consists of parallel activities carried out by the same object. For example, a customer entering a bank may decide that (s)he will wait in line for no longer than two minutes. At the end of that time, if the customer has not been served, (s)he will renege (exit the system without having been served.)

At first glance, the distinction between large-scale and local parallelism might appear to be somewhat arbitrary. What's large-scale, and what's local? In SLX, a precise distinction is made based on the SLX verbs which are used. Objects are activated by using the *activate* verb. In most cases activate is used in conjunction with new:

```
activate new customer;
```

The above statement creates a new customer object, creates a puck for it and places the puck in the active puck list, poised to execute the first statement in the customer object's actions property. The puck which executes the "activate new" statement continues executing. The new customer will compete with other pucks (according to their respective priorities) after the current puck has gone as far as it can in the actions property at the current instant of simulated time.

Local parallelism is described by using the *fork* statement:

```
fork
  {
  offspring actions
  }
parent
  {
  parent actions
  }
```

Execution of the fork statement creates an additional puck for the currently active object. The newly created puck is placed in the active puck list, poised to execute the offspring actions clause of the fork statement. The parent puck executes the optional parent actions clause. Unless otherwise specified, both pucks continue execution with the next statement following the fork statement. An example of the fork statement is given in Section 2.4.

### 2.3 Time Advance

Time advance is provided by the advance statement, modeled on the ADVANCE block of GPSS/H, e.g.,

```
advance rv_expo(stream2, 10.0);
```

The advance statement removes the puck from the active puck list and places it on the future event list, scheduled to resume execution after the specified time delay has taken place.

### 2.4 Control Variables & Wait Until

In SLX, state-conditioned delays are modeled using *control* variables and the *wait until* statement. The keyword "control" is used as a prefix on SLX variable declarations:

```
control integer count;
control boolean repair_completed;
```

The "control" keyword tells the SLX compiler that at each point at which the value of the control variable is changed, a check must be made to see whether any pucks in the model are currently waiting for the variable to attain a particular value or range of values. Such waits are described using the *wait until* statement:

```
wait until (count > 10);
wait until (repair_completed);
```

Compound conditions are allowed as well:

```
wait until (count >= 10
  or repair_completed
  and not repairman_busy);
```

SLX also supports *indefinite* (user-managed) waits. Three steps are required to implement an indefinite wait. First, the puck which is going to wait must be made accessible to other pucks. This is usually done by placing the puck into a set. Second, the puck executes a wait statement with no "until" clause. Finally, at a

subsequent point in simulated time, another puck executes a *reactivate* statement to reactivate the waiting puck.

Let us consider an example which illustrates the use of the fork statement in conjunction with *wait until*. Assume that customer objects flowing through a model reach a point where they are willing to wait a maximum of two minutes for service. If they are not served within two minutes, they exit the system; i.e., they renege.

```
object customer
  {
  control boolean reneged;
  actions
    {
    ...
    fork
      {
      advance 2.0;    // max wait time
      reneged = TRUE;
      terminate;
      }
    parent
      {
      wait until (reneged
         or server available);
      if (reneged)
      terminate;
      }
    ...
    }
  };
```

In the above example, a Boolean control variable is used within the customer object for communicating "renege" status between two pucks which share the same object. At the point at which the customer begins waiting for service, it forks, creating a second puck. The offspring puck executes the logic enclosed in braces immediately following the fork statement. The original puck executes only the logic contained within braces following the "parent" clause. The offspring puck undergoes a two-minute delay, sets reneged to TRUE, and terminates itself. The parent puck waits for either the server to become available or for the two minutes to elapse. When it comes out of the `wait until`, it must distinguish which of these two possibilities has taken place. If the two minutes have elapsed, reneged will be TRUE, and the parent puck will terminate itself. If not, the parent will continue executing.

The sharing of a single data area makes communication between the two pucks trivial. The "reneged" variable shared by the two pucks is all that is needed to accomplish the communication required in this example. Note that if there are multiple customers active at a given time, each customer will have its own data area, so the "reneged" status for one customer cannot be confused with that of another.

In many simulation languages, operations such as reneging are difficult to implement. Because of this, languages sometimes include operations such as reneging as built-in features. Unfortunately if the language designer's concept of reneging does not exactly match your requirements, you're stuck. In SLX, carefully designed rock-bottom primitives allow you to build your own capabilities if none of the ones provided by others meet your needs.

### 2.5 Sets

SLX includes the capability for defining and manipulating sets of objects. Sets can be homogeneous (comprised of objects of a single type) or universal (comprised of objects of arbitrary types.) Homogeneous sets can be ranked in ascending or descending order on one or more attributes of objects of the type comprising the set. Some examples of set definitions follow:

```
set(widget) ranked FIFO  fifo_set;
set(widget) ranked LIFO  lifo_set;;
set(*)  universal_set;
set(job) ranked(ascending due_date,
descending priority)   job_set;
```

An iteration construct is provided for looping through the members of a set:

```
pointer(widget) w;
for (w = each widget in fifo_set)
  {
  ...
  }
```

If the subject set is a universal set, `for each` selects only objects of the specified type, skipping over any objects which are of other types. Arbitrarily complex forms of iteration can be built from lower-level first, last, successor, and predecessor primitives:

```
w = first widget in widget_set;
w = last widget in widget_set;
w = successor(w) in widget_set;
w = predecessor(w) in widget_set;
```

Using these primitives, *any* form of set iteration can be achieved.

## 3 EXTENSIBILITY FEATURES

SLX was designed to be an extensible platform on which a wide variety of higher level simulation applications could be built. In this section we will briefly present an example of how the extensibility mechanisms can be used to build new features out of old ones.

Suppose we wish to build a simple telephone book. Furthermore, assume that each entry in the book contains only a first name, a last name, and a telephone number. An entry in the phone book can be described as an SLX object:

```
object  book_entry
  {
  fstring(20)     first_name,
                  last_name;

  int             phone_number;
  }
```

Suppose that we wish to retrieve entries from the phone book both by name and by number. We could construct two SLX sets, one ranked by name and one ranked by number:

```
set(book_entry)ranked(ascending
  last_name, ascending first_name)
    phone_book;

set(book_entry) ranked (ascending
  phone_number)
    reverse_phone_book;
```

The name-sorted phone book could be printed as follows:

```
pointer(book_entry) name;
for (name = each book_entry in
  phone_book)
    print( name -> last_name,
           name -> first_name,
           name -> phone_number)

"*****, *****: *******\n":
```

Of course, the printing of phone books is best left to the telephone company. For the rest of us, the most common use of the phone book is to look up numbers. For the police, a relatively common use is to look up the name associated with a number. The SLX kernel contains no "look up" primitive; however, one can be constructed easily from existing language features. The following approach can be used to look up a phone number, given first and last names. First, copy the first and last names into a dummy book_entry object. We call this object query_book_entry; Next, place the query_book_entry object into phone_book. Since phone_book is ranked by ascending last and first name, and entries with identical keys are inserted into a set in FIFO order, we know that query_book_entry will be placed into phone_book immediately following the entry we wish to look up. Thus, the desired entry will be the predecessor of query_book_entry in phone_book. If the name we are looking up is not in the phone book, either the predecessor of query_book_entry will be NULL, or it will contain the wrong names. These conditions are easily detected. Finally, the query_book_entry can be removed from phone_book.

While each of the steps outlined above is straightforward, we'd like to have a less cumbersome way of issuing queries. SLX provides a statement definition facility which allows us to construct a "retrieve" statement. The definition of our retrieve statement is shown in Figure 1. The first line of the definition is a prototype which specifies the components of the retrieve statement. Names preceded by a pound sign ("#") represent components that are supplied by the user for each use of the statement. Braces ("{}") are used to enclose a group of specifications. The notation ",..." indicates that the immediately preceding component can be repeated as many times as desired, with repetitions separated by commas. The "@" in front of the "from" keyword tells SLX to ignore the usual meaning of "from" and treat it as a keyword of the retrieve statement. ("from" is a reserved word in SLX.)

The definition section specifies the mapping of the retrieve statement into lower-level SLX statements. Within the definition section, the *expand* statement is used to specify the lower-level SLX code that is to carry out the retrieval operation. The expand statement specifies one or more lines of output which is injected into the SLX compiler's input stream. A list of expressions can be supplied to be edited into the generated lines of output. Within an output line, groups of adjacent "#" symbols are replaced by edited values.

With one very important exception, this approach is similar to the use of *macros* in many languages. In most languages, the statements which are available to specify the internal logic of a macro are either very limited and use a syntax different from the host language, or they comprise a comparatively weak subset of the host language. In SLX, the "macro language" is SLX itself. Only a handful of statements are excluded from use within an SLX statement definition. For example, simulation constructs such as *wait until* or *advance* have no meaning during compilation of a program. Apart from these obvious restrictions, most of the rest of the SLX language can be used. For example, it is even possible to read a file as part of the process of expanding a statement!

The example in Figure 1 makes use of a local integer variable, i, which is used to iterate through the list of comma-delimited "#key = #value" items. The iteration is terminated when an empty value of #key[i] is encountered. This is an SLX convention. If a list of N items is provided, #key[1...N] will have non-empty values, and #key[i] will be empty for i > N.

A sample invocation of the retrieve statement and its expansion is shown in Figure 2. This example illustrates the power of SLX's statement definition facility. The retrieve statements are easy to read. If you didn't know that the retrieve statement was built using the statement definition facility, you would probably think that it was a built-in SLX statement. This example illustrates the extensibility of SLX. It's very easy to reshape the language by adding new statements which are specific to the kinds of problems you are working with. Whether you need a capability which is not present in basic SLX, or whether you just want a more expressive way of specifying complex logic in a more compact form, the statement definition facility can be of great use.

```
statement retrieve #ptr = #otype ( { #key = #value },... ) @from #set ;
    definition
        {
        int     i;

        expand                              "{\n";

        for (i=1; #key[i] != ""; i++)
                expand(#otype, #key[i], #value[i])

                                            "query_#.# = #;\n";

        expand(#otype, #set)                "place &query_# into #;\n";

        expand (#ptr, #otype, #set)         "# = predecessor(&query_#) in #;\n";

        expand(#otype, #set)                "remove &query_# from #;\n";

        expand(#ptr, #ptr, #key[1], #value[1])
                                            "if (# != NULL)\n  if (# -> # != #";

        for (i=2; #key[i] != ""; i++)
                expand(#ptr, #key[i], #value[i])

                                            " or # -> # != (#)";

        expand(#ptr)                        ")\n     # = NULL;\n";
        expand                              "}\n";
        }
```

Figure 1: The Definition of a "Retrieve" Statement

```
retrieve e = book_entry(last_name=lname, first_name=fname) from phone_book

    {
    query_book_entry.last_name = lname;
    query_book_entry.first_name = fname;
    place &query_book_entry into phone_book;
    e = predecessor(&query_book_entry) in phone_book;
    remove &query_book_entry from phone_book;
    if (e != NULL)
      if (e -> last_name != lname or e -> first_name != (fname))
         e = NULL;
    }
```

Figure 2: A Sample Expansion of the "Retrieve" Statement

## 4 EXPERIENCE WITH SLX TESTERS

As of this writing SLX has been intensively used by a select group of testers around the world. Interacting with the testers has resulted in a number of improvements to SLX and has yielded some interesting insights into how SLX is (and will be) used. The projects tackled by SLX testers have all been fairly intense efforts. Included among these have been a half-dozen masters degree thesis projects and one very large scale, real-world transportation model.

Two modest surprises have come out of the testing. First, the extent to which users have made direct use of SLX kernel-level features has exceeded our expectations. Perhaps biased by our years of experience with GPSS/H, we had expected that most users would prefer to use higher-level features, only occasionally resorting to kernel-level features. Virtually all the testers have made heavy use of low-level scheduling primitives such as wait until and fork. A second modest surprise has been the extent to which the testers have exploited SLX's statement-definition capabilities. We had envisioned the statement definition capability as being of interest primarily to builders of higher-level simulation packages, but nearly everyone who has touched SLX has made use of its statement definition facility.

Taken together, these two surprises demonstrate that we have achieved our goal of extensibility. SLX is easily adapted to different purposes by different users. With the ability to incorporate new statements into the language, the extended language used by the developer of a package for manufacturing applications is quite different from the extended language used by someone tackling transportation problems. Yet the two are built on a common foundation, and the underlying software is hardened and tempered by exposure to widely differing usage patterns.

## 5 TEACHING SLX

The architecture of SLX has potentially profound implications for teaching simulation. The usual approach to teaching simulation is to "dive in" at an intermediate level by providing an easily understood collection of building blocks and exploring some well-motivated examples. Students of simulation who tackle real-world applications sooner or later reach a point at which they have to go back and build a foundation under their knowledge; i.e., they have to learn how things really work. Depending on exactly when the foundation-building process takes place, students may have already developed usage patterns which ignore some of a language's capabilities and misuse others. For example, self-taught users of GPSS/H will almost always favor an "active-object, passive-server" world-view, even though the language is quite capable of expressing an "active-server, passive object" world-view. For users of very high-level simulation packages, especially graphically based model-builders, the foundation-building may *never* take place. Whether this is good or bad is a matter of religion. Advocates of the very high-level approach think this is good, while their more conservative counterparts are appalled by the danger of doing too much with too little knowledge.

In SLX, the number of kernel constructs which directly support simulation is very small. Depending on what one counts as a simulation feature, the number ranges from roughly 8 to 12. Our experience with GPSS/H has proven that this is a small enough number of building blocks for beginners to readily absorb. For example, we have seen many times that so-called "9-block GPSS/H" is easily mastered and quite powerful.

However, even with 9-block GPSS/H, students quickly reach a point at which foundation-building is necessary. With SLX, a bottom-up approach is feasible. For example, consider modeling a barbershop, a traditional introductory one-line, single-server queuing model. In a beginner's model, the barbershop runs from 9:00-5:00, at which time it summarily shuts down, ignoring the customer (if any) who is in the barber chair at that time and ignoring customers (if any) in the queue. In a second model, more realistic shutdown conditions can be implemented. At 5:00 the door to the shop is closed, and the barber does not leave until the current customer and all customers in the queue at 5:00 have been served. In SLX, this condition is easily expressed as a compound "wait until" condition, e.g., "wait until (queue empty and server idle)." Thus, SLX's wait until feature is well-motivated and easily understood at a very early stage of model building. In SLX, wait until is the foundation of *all* forms of state-based events. Thus mastery of wait until yields enormous benefits.

SLX kernel-level simulation primitives are *exposed*, i.e., they can be used *directly*. In most simulation software, primitives are bound into impenetrable higher-level features. For example, in GPSS/H there are at least five building blocks which internally utilize the equivalent of wait until. Some of these blocks have many external variations. Thus, students of GPSS/H must master the external variations *and* learn how the underlying `wait until` mechanism works. In SLX, it's easier to learn the general mechanism first. Wait until is both an SLX primitive and a fundamental modeling concept. Thus, by teaching/learning `wait until`, we can kill two birds with one stone.

As of this writing, an introductory SLX textbook is under development. A preliminary (at least) edition of the book will be available by the time of the 1996 Winter Simulation Conference. The book will use the approach outlined above for simultaneously exposing fundamental simulation concepts and their implementation in SLX. Given the enormous popularity of graphically-based, fill-in-the-blanks modeling tools, our approach to teaching SLX entails a degree of risk. We're going back to teaching the basics at a time when much of the rest of world is moving in the opposite direction. Our approach is motivated by a year's experience with our SLX testers and over twenty years' experience in helping users learn to use our simulation software. Our experience is biased by years of exposure to people who still believe it's important to know what they're doing.

## 6 CONCLUSIONS

SLX is a well-conceived, layered simulation system. Users of the upper layers can ignore lower layers. However, if their requirements are not met at a given level, they can move down one or more levels, without exerting extraordinary effort and without losing protection against potentially disastrous errors. Developers, who are used to working down among the lower layers, have at their disposal powerful extensibility mechanisms for building higher layers for use by themselves or others. The efficacy of the approaches offered by SLX has been demonstrated through intensive use. The benefits for teaching simulation derived from being able to simultaneously expose language primitives and fundamental modeling concepts remain to be demonstrated as of this writing. However, we are excited by SLX's great potential.

## REFERENCES

Henriksen, J.O. 1995. An Introduction to SLX. In *Proceedings of the 1995 Winter Simulation Conference*, ed. C. Alexopoulos. 502-509. Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

Henriksen, J.O., and R.C. Crain. 1996. *GPSS/H reference manual*, fourth edition. Annandale, VA: Wolverine Software Corporation.

## AUTHOR BIOGRAPHY

**JAMES O. HENRIKSEN** is the president of Wolverine Software Corporation. He is a frequent contributor to the literature on simulation and has presented many papers at the Winter Simulation Conference. Mr. Henriksen served as the Business Chairman of the 1981 Winter Simulation Conference and as the General Chairman of the 1986 Winter Simulation Conference. He has also served on the Board of Directors of the conference as the ACM/SIGSIM representative.