AN INTRODUCTION TO SLX™

James O. Henriksen

Wolverine Software Corporation
3131 Mounte Vernon Avenue
Alexandria, VA 22305-2640, U.S.A.

ABSTRACT

SLX is Wolverine Software's "next generation" simulation language. SLX builds on the strengths of Wolverine's GPSS/H (Crain, 1997). 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 focuses on three key features of SLX. The power of these features is illustrated by presenting an SLX solution to a small, but moderately difficult simulation/animation problem. An earlier paper (Henriksen 1996) provides a broader overview of SLX for readers who are already familiar with simulation, and (Henriksen 1995) presents key concepts of the architecture of SLX.

1 INTRODUCTION

This paper presents three key features of SLX: (1) SLX's ability to describe parallelism in systems being modeled, (2) SLX's generalized *wait until* mechanism, and (3) SLX's statement definition mechanism, which allows the introduction of new, user-defined statements into the SLX language. In the section which follows, a small, but moderately complex conveyor modeling problem is presented. In subsequent
sections, the three aforementioned features are presented in detail, and the layered solution to this problem is discussed. Along the way, representative components of each layer are presented. The lowest layers include subroutines written in SLX which serve as computational building blocks for intermediate layers. The highest layer is built by defining new statements for describing the operation of conveyor systems. The end result is a small conveyor-modeling package which is dialect of SLX. This package is suitable for use by "end users" as it stands. However, the beauty of the layered SLX implementation is that users with specific needs not addressed by the general-purpose package can go down one or more levels, to replace, modify, or add to the conveyor modeling features which have been implemented.

2 A SAMPLE PROBLEM

Figure 1 depicts a small, hypothetical conveyor system This system is an extension of a system for which a GPSS/H solution was presented in an earlier paper (Henriksen 1986). A brief description of the system follows.

Four "pickers" are assigned to adjacent 25-foot sections of a 100-foot wide storage area. The pickers are responsible for picking cartons from storage and placing the cartons on a conveyor belt. The orders for each picker, are uniformly distributed along a 25-foot storage area. Cartons are of 11-inch, 17-inch, and 23-inch lengths, uniformly distributed. The time required to pick ranges from zero to 10 seconds, depending on how far a given order is from the previous order. (Zero distance implies zero time, and the maximal 25-foot distance implies a 10-second picking time.) Pickers are initially positioned in the middle of their 25-foot sections.)

Once a picker has picked a carton from storage, (s)he carries the carton along a path perpendicular to the conveyor belt. When a picker reaches the conveyor belt (s)he must wait for a an available space to come by which is 6 inches wider than the carton on either side. The conveyor belt moves at a speed of 1 foot/second, unless it is stopped (discussed below). Pickers are lazy. If space is not available, they wait for it to appear, rather than walking to the left or right, trying to find available space. If space is available, a carton can be placed on the conveyor, whether or not the conveyor is stopped, and the picker can move on to his/her next order.

The conveyor belt extends 10 feet past the end of the storage area; i.e., it is a 110-foot conveyor. The conveyor belt connects to a 30-foot section of accumulating conveyor which also moves at a speed of one foot/second. Cartons queue up at the downstream

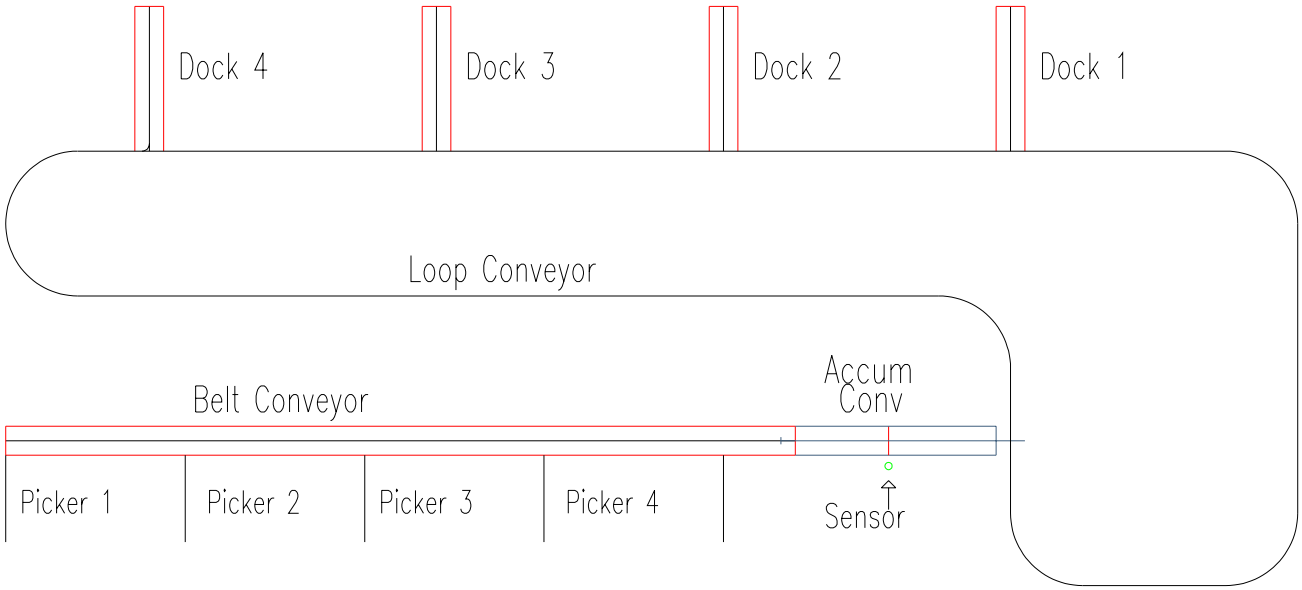

Figure 1: A Hypothetical Conveyor System

end of the accumulating conveyor, awaiting placement on a circulating loop conveyor. A light beam sensor is positioned at the midpoint of the accumulating conveyor. Each time a carton passes the sensor the beam is broken. If the beam stays continuously broken for a 3-second period, the 110-foot conveyor belt is stopped until the beam is once again visible.

At the end of the accumulating conveyor, cartons are placed on a loop conveyor which circulates at a speed of 2 feet/second. Placing a carton requires a 6-foot free space on the loop conveyor (3 feet either side of where the front edge of the carton is placed). A 2.5-second cycle time is incurred to place a carton on the loop; i.e., remaining cartons on the accumulating conveyor do not advance until 2.5 seconds after a carton is placed on the loop.

Each carton is assigned a destination, uniformly distributed among four loading docks. The loop conveyor has a length of 454.2477 feet, and the four loading docks are located at offsets 147.24, 187.24, 227.24, and 267.24 along the loop. When a carton reaches its destination, subject to a limitation described below, it is placed on a 20-foot conveyor which moves at 6 inches/second. When a carton reaches the end of a loading dock conveyor, it vanishes from our hypothetical system. The transfer of a carton from the loop to a loading dock conveyor requires a minimum 10-second cycle time. If a second carton reaches its target loading dock sooner than 10 seconds after its predecessor has reached the same destination, the second carton must go around the loop and try again. It is possible for a carton to circulate 2, 3, 4, or more times around the loop.

3 MODELING PARALLELISM IN SLX

3.1 Active Objects and Pucks

In SLX, two kinds of objects are used to represent components of systems being modeled. *Passive* objects are used for modeling entities which have no "executable" behavior. In our conveyor model, cartons are modeled as passive objects acted upon by other objects. (For those readers familiar with C, passive objects are very much like C structs.) *Active* objects have executable behavior patterns specified in an *actions* property of the object's class definition. In our conveyor model, active objects are used to model pickers and "transfer managers," which supervise the transfer of cartons from one conveyor to another.

Objects are created by using the *new* operator, which returns a pointer to the newly created object. When an *activate* operator is applied to a pointer to an object, a *puck* (defined below) is created for the object and placed on the Current Events Chain; i.e., the puck is placed in a ready-to-execute state. The new and activate operators are almost always used in a single statement:

```
activate new Picker(25.0, 50.0);
```

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. The SLX simulator can be regarded as a puck manager.

3.2 Active Objects and Pucks vs. Transactions

The original version of GPSS introduced the transaction-flow modeling paradigm to the world in 1962. In the transaction-flow world view, attention is focused on units of traffic, called transactions, which flow through a system, competing for system resources. In the 35-year period since GPSS was introduced, a number of other languages have implemented variations of the transaction-flow world view. Implementation of this world view, and the terminology used to describe it vary widely (See (Schriber and Brunner 1997)).

In traditional transaction-flow languages, a transaction contains two types of data, user-defined data particular to the unit of traffic, and "scheduling" data, needed to keep track of the state and "location" of the unit of traffic in a model. Figure 2 shows this distinction. In a GPSS model of our conveyor system, a transaction representing a picker would have attributes such as left edge of picking zone, right edge, current position, etc. Scheduling data would include priority, next scheduled "move time", next model statement to be executed, etc. Scheduling data includes values which can be modified by a program, e.g., transaction priority, and other values which are "internal" values maintained by run-time support routines for the simulation language. All user-defined transaction data can be both read and written by user code.

Scheduling Data	User-Defined Attribute Data

Figure 2: The Structure of a Transaction

In SLX the functionality of a transaction is broken down into independent lower-level components, and there are no transactions, *per se*. The role of a transaction's user-defined data is played by an SLX user-defined *object class*. The role of a transaction's scheduling data is played by an SLX *puck*. The statements which are executed by the object are contained in the *actions property* of the object's class and any lower-level procedures invoked by the actions property.

3.3 Inter-Object and Intra-Object Parallelism

In SLX, parallelism can be modeled in two ways: as interactions among objects (inter-object parallelism) and as multiple actions performed on behalf of the same object (intra-object parallelism.) The SLX conveyor model uses both methods. For example, the four picker objects interact (albeit indirectly) by virtue of the fact that they compete for space on the belt conveyor. Resource contention of this form is characteristic of transaction-flow languages. Intra-object parallelism is achieved by creating one or more additional pucks for an active object. This is accomplished by means of a *fork* statement. The following code is used within a loading dock active object to model the concurrent elapsing of the 10-second cycle time before another carton can be taken off the loop conveyor and placed on the loading dock, and the elapsing of the 40-second time it takes a carton to reach the end of the loading dock conveyor and exit the system:

```
fork
    {
    advance 40.0;          // box travel time;

    "Remove load from loading dock."

    terminate;
    }

advance 10.0;              // Cycle time
```

In the above example, the fork statement creates a second puck for the loading dock object. The offspring puck is placed on the Current Events Chain, poised to execute the actions within the braces ("{…}") following the fork statement. The parent puck continues its execution with the advance 10.0 statement which follows. Thus the first 10 seconds of the offspring puck's 40-second time delay overlap with the parent puck's 10-second time delay.

Most transaction-flow simulation languages offer only inter-object parallelism. Most offer some form of "cloning" operation which is superficially similar to SLX's fork statement. When such an operation is performed, a new transaction is created. The new transaction, by definition, has its own scheduling data, and usually the user-defined attributes of the parent transaction are copied into the offspring (clone). Refer back to figure 2. SLX's fork statement creates a new puck (scheduling data *only*) which shares the user-defined attributes with other pucks.

If a language has only a transaction-cloning verb, and no fork verb, certain types of system behavior are more difficult (although certainly not impossible) to model. Consider, for example, GPSS/H's SPLIT block, which creates a clone of an entire transaction. If multiple GPSS/H transactions need to share a single copy of data describing a component of a system, the data must be stored in global variables. (In GPSS/H, transactions can easily change their own attributes, but

changing the attributes of other transactions is difficult. Thus, storing the shared data in any given transaction is impractical.) If more than one such component exists in the system being modeled, separate collections of shared global variables must be used, one collection for each such component. If the collection of system components requiring such representation does not change during model execution, the shared data can be statically allocated. However, if the collection of components changes during model execution, some form of dynamic data management must be implemented by the modeler, since GPSS/H global variables are statically allocated at the start of model execution; i.e., they cannot be created and destroyed during model execution.

The fork statement, which enables intra-object parallelism, is an extremely handy modeling tool. Even in the simple example shown above, it's very helpful. When the parent puck completes its 10-second delay, it can continue its task (waiting for, and reacting to the presence of another carton on the loop) without having to "worry" about the offspring puck's disposing of the previous carton. In more complex modeling situations, intra-object parallelism can be indispensable. Suppose a class of objects has a lot of local data (many attributes), that members of the class are dynamically created and destroyed during model execution, and that multiple instances of the object exist at any given time. Even in such complex circumstances, the use of multiple pucks offers easy shared access to object attributes among all the pucks which belong to any given instance of the object, while preventing access by pucks which belong to a different instance.

4 SLX's GENERALIZED WAIT UNTIL

As units of traffic flow through a model, they are subject to two forms of delay, scheduled delays, and state-based delays. In SLX, state-based 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.

Wait until expressions can include time-based conditions. **time** is a reserved word in SLX, denoting the simulator clock. Let us consider the six possible forms of time-based wait until expressions that could be used. (Since **time** can occur either to the left or right of a comparison operator, there are six mirror-image forms with **time** on the right, which are not shown.)

```
wait until (time <  expression);
wait until (time <= expression);
wait until (time == expression);
wait until (time !=  expression);
wait until (time >  expression);
wait until (time >= expression);
```

The first two forms are not allowed, because the simulation clock cannot run backwards. If such expressions were allowed, once they became false, they could never become true, because time can only increase during model execution. This leaves four cases. "Time == expression" is handled by scheduling a hidden event to occur at the specified time. Using this form is functionally equivalent to using an *advance* statement (scheduled time delay), although it incurs a little additional overhead for going through the wait until mechanism. If the value of time has already passed the value specified in the expression, an execution error occurs, since the clock cannot run backward, which would be required to satisfy the condition. "Time != expression" is handled by treating time as a control variable. This leaves two cases of interest. "Time >= expression" is treated in the same manner as "time == expression", except that no error occurs if the value of time has already passed that specified in the expression. "Time > expression" is treated in two stages. First, a hidden event is scheduled to occur at the specified time. Then, when this time is reached, the wait until is completed by treating time as a control variable.

Wait until expressions can combine time-based terms with other terms. In our conveyor model, this feature is

very heavily used. Let us consider a simplified version of a subroutine designed to move a carton from point A to point B along a conveyor. In our model, a "LoadStatus" object is created to describe each carton on a conveyor. The LoadStatus object contains a control integer variable, called ChangeCount. Each time another object performs an action which affects a given LoadStatus object, the object is responsible for incrementing the LoadStatus's ChangeCount. For example, if a LoadStatus object is inserted in front of another LoadStatus object, the ChangeCount of the LoadStatus object in front of which the new LoadStatus object was inserted must be incremented. In practice, compliance with this discipline is easy. In virtually all cases when a change occurs to a given LoadStatus object, only the upstream neighbor LoadStatus object must be notified.

The following is a sketch of the heart of the subroutine to move LoadStatus objects from point A to point B:

```
pointer(LoadStatus)     Load;

"set Load's current position to B."

forever
        {
        wait until (ConveyorSpeed > 0.0);

        Distance = B - CurrentPosition(Load);
        ArrivalTime = Distance / ConveyorSpeed;
        LocalCount = Load -> ChangeCount;

        wait until (time == ArrivalTime
        or Load -> ChangeCount != LocalCount);

        if (Load -> ChangeCount == LocalCount)
                break;   // exit the forever loop
        }
```

The code shown above goes around and around a forever loop until its LoadStatus object reaches point B. Let us consider the "outside" forces that can influence the transit of a LoadStatus object within this loop. The speed of the conveyor on which the LoadStatus object is being transported can be changed. When the speed of a conveyor is changed, the ChangeCount attributes of all loads on the conveyor are incremented. The forever loop contains a wait until which waits until the speed of the conveyor is greater than zero. This prevents division by zero in the body of the loop. Each time through the loop, a scheduled arrival time is computed for the load, given its current position (returned by the CurrentPosition function). A wait until is executed waiting for either (1) that time to be reached or (2) some change affecting the LoadStatus object to occur. The code is "bullet-proof." The only discipline that must be strictly adhered to is to increment the ChangeCount of any LoadStatus objects affected by system state changes.

5 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 present a small example of how SLX's statement definition facility was used to build our conveyor modeling software.

In the process of developing the conveyor modeling software, we eventually reached a point at which a hierarchical collection of objects and subroutines had been constructed, and the collection was capable of nicely modeling systems of belt conveyors, accumulating conveyors, loop conveyors, and sensors. We could have stopped at this point, documented the objects and subroutines, and said to prospective users, "if you want to model conveyors, these are the subroutines you need, and this is how you use them." However, we wanted to take things a step further, and build an easier-to-use, statement-oriented interface on top of the collection of objects and subroutines. The complete collection of statements is described in section 6. In the paragraphs which follow, we will consider one such statement, CVR_Send.

The CVR_Send statement is used to send a LoadStatus object to a destination on a conveyor. If no destination is specified, the downstream end of the conveyor is assumed. The puck which executes a CVR_Send statement does not wait for the LoadStatus object to reach its destination; however, it can subsequently test for and/or wait for the LoadStatus object to arrive at its destination. An alternative statement, CVR_Ride, does wait for a LoadStatus object to reach its destination. CVR_Send is an asynchronous form, and CVR_Ride is a synchronous form of LoadStatus object transport. The following are examples of CVR_Send:

```
pointer(LoadStatus)     Load;

CVR_Send Load to 47.5;
CVR_Send Load;
```

Note that the conveyor on which the LoadStatus object is to be sent is not specified. This is because the LoadStatus object must have been previously placed on the conveyor. In fact, the placing of a carton onto a conveyor creates the LoadStatus object which is used in a CVR_Send statement.

The definition of the CVR_Send statement is shown in Figure 3. The first line of the definition is a prototype

which specifies the components of the CVR_Send statement. Names preceded by a pound sign ("#") represent components that are supplied by the user for each use of the statement. Brackets ("[]") are used to enclose a group of optional specifications. The "@" in front of the "to" keyword tells SLX to ignore the usual meaning of "to" and treat it as a keyword of the CVR_Send statement. ("to" is a reserved word in SLX.)

The definition section specifies the mapping of the CVR_Send 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! In CVR_Send, the form of expansion used depends on whether or not a "to" clause is supplied. In either case, a call of ConveyLoad, a lower-level run-time routine, is generated.

```
statement CVR_Send          #Load [@to #Destination] ;
 definition
    if (#Destination != "")
        expand (#Load, #Destination)     "ConveyLoad(#, #);\n";
    else
        expand (#Load, #Load)        "ConveyLoad(#, (#) -> LoadConveyor -> ConveyorLength);\n";
```

Figure 3: The CVR_Send Statement Definition

6 THE LAYERED APPROACH

Our system for modeling conveyor systems was constructed as follows:

A. Data structures were designed for representing conveyors, sensors, and loads. While the data structures are unremarkable, one feature should be pointed out. Early on, it became apparent that the *status* of objects being conveyed should be stored in separate LoadStatus objects which were independent from, but connected to, the "user-level" objects being conveyed (cartons in our little example). This approach offered several advantages. First, it allowed the hiding of lower-level details from higher-level code. Second, it allowed the writing of lower-level routines in a load-independent style. Third, it allowed a many-to-one relationship between LoadStatus objects and user-level objects being conveyed. When an object is transferred from one conveyor to another, if the transfer is not instantaneous, during the period in which the transfer takes place, LoadStatus objects exist for both conveyors. Thus, if the second conveyor is stopped before an object has moved completely off the first conveyor, appropriate actions can be taken; e.g., the first conveyor can be stopped, if necessary.

B. A number of low-level functions were written to perform object position accounting, e.g.,

CurrentPosition(Load).

C. Functions were written to scan conveyor data structures to find loads that were located within *zones* of a conveyor, e.g.,

FindFirstLoadInZone and
FindLastLoadInZone.

D. Four functions were developed to wait for the following zone statuses to arise:

WaitForZoneFull
WaitForZoneNotFull
WaitForZoneEmpty
WaitForZoneNotEmpty

WaitForZoneFull returns a pointer to the LoadStatus object for the load which is furthest downstream in a zone, and WaitForZoneEmpty returns a pointer to the LoadStatus object for the next load which is scheduled to enter a zone.

The availability of WaitForZoneEmpty allows trivial implementation of random placement of loads. One needs only to specify a conveyor zone which allows sufficient clearance before and after a load, and issue a call to WaitForZoneEmpty. The availability of WaitForZoneFull allows trivial implementation of sensors. A sensor can be modeled as a very narrow zone, corresponding to a light beam. WaitForZoneFull can be used to wait for the beam to be broken, and since this function returns a pointer to the furthest downstream load in the zone, the identity of the load which has broken the beam can be ascertained.

E. The following statements were defined for modeling conveyors:

```
CVR_Conveyor
CVR_Sensor
CVR_LoadClass
CVR_ConveyorSpeed(c) = s;
CVR_LoadSpeed(l) = s;
CVR_Place … on
CVR_Remove … from
CVR_Send
CVR_Ride
```

The first three statements are used to define elements of a conveyor system. The next two are used for setting load and conveyor speeds. The final four are used for placing, removing, and transporting loads.

7 TAKING THINGS A STEP FURTHER

The sample system was animated, using Proof Animation (Henriksen 1997). The animation was accomplished by using statements built using the SLX statement feature. Animating the model added 14 statements to the model. This number could have been smaller, had we chosen to imbed animation statements within our various CVR_… statements. This approach was not taken, because we wanted to maintain a degree of independence between animation and simulation. If we had "hard-wired" the animation capabilities into our conveyor routines, tailoring the animation to specific needs would have become more difficult.

One of the users of SLX and Proof Animation has constructed SLX code which is capable of reading Proof Animation layout files and extracting information about path (conveyor) lengths, sensor positions, etc. This allows the use of Proof Animation as a visually-based design tool for conveyor systems. If a conveyor needs to be lengthened, or sensor positions need to be changed, it's much easier to do this sort of thing visually and use a computerized interface for automatically incorporating system geometry changes in a model. Manually entering data such as conveyor lengths and sensor positions is a tedious, error-prone task.

8 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. The conveyor modeling package we have discussed in this paper demonstrates the efficacy of the SLX approach. The prototype system developed is itself applicable to a wide range of conveyor systems. More importantly, the layered fashion in which it was constructed allows the convenience of a high-level package and the confidence that if a new application falls outside the built-in capabilities of the software, the software can be readily extended by dropping down a layer or two.

REFERENCES

Crain, R.C. 1997. Simulation Using GPSS/H. In *Proceedings of the 1997 Winter Simulation Conference*, eds. S. Andradóttir, K. Healy, D. Withers, B. Nelson.

Henriksen, J.O. 1997. The Power and Performance of Proof Animation. In *Proceedings of the 1997 Winter Simulation Conference*, eds. S. Andradóttir, K. Healy, D. Withers, B. Nelson.

Henriksen, J.O. 1996. An Introduction to SLX. In *Proceedings of the 1996 Winter Simulation Conference*, eds. J. Charnes, D. Moore, D. Brunner, J. Swain. 468-475. Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

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.

Henriksen, J.O. 1986. You Can't Beat the Clock. In *Proceedings of the 1986 Winter Simulation Conference*, eds. J. Wilson, S. Roberts, J. Henriksen.

713-726. Institute of Electrical and Electronics Engineers, Atlanta, Georgia.

Schriber, T.J., and D.T. Brunner. 1997. Inside Simulation Software: How It Works and Why It Matters. In *Proceedings of the 1997 Winter Simulation Conference*, eds. S. Andradóttir, K. Healy, D. Withers, B. Nelson.

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.