SLX - PYRAMID POWER

James O. Henriksen

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

ABSTRACT

SLX is Wolverine Software's "next generation" simulation language. It provides powerful simulation capabilities in a modern framework. SLX is structured as a multiplicity of layers, ranging from its C-like 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. SLX contains powerful extensibility mechanisms for building new layers atop old ones. (The X in SLX stands for eXtensibility.) SLX also contains innovative features for coupling SLX to other languages and packages. This paper presents an overview of SLX. Earlier papers (Henriksen 1997, 1998) presented the development of a conveyor modeling package in SLX, and example of how SLX has been coupled with other software, respectively.

1 INTRODUCTION

The most important characteristic of SLX is its layered, pyramidal, architecture, shown in Figure 1.

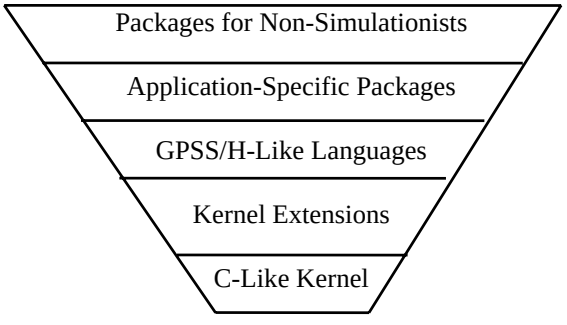

Figure 1: The SLX Pryramid

Traditional language-based simulation tools fall in the middle of the SLX pyramidal hierarchy. The efficacy of SLX's layered approach is hinges on four key factors:

A. SLX's layers are well-conceived. In each layer, we have taken a minimalist-generalist approach. Consider the design of SLX's kernel. We went to great lengths to minimize the "footprint" of the kernel. As a result of our approach, the SLX kernel is surprisingly small collection of precisely-defined, very general primitives which can support a wide variety of higher-level modeling approaches. For example, SLX's kernel-level *wait until* statement allows easy specification of state-based events, e.g., "wait until State A and State B or State C." State-based events are the foundation of a variety of world views, e.g., transaction flow, process interaction, activity scan, and Petri nets.

B. SLX's layers are properly separated. Many other modeling tools provide multiple layers, but often there are wide gulfs between the layers, leading to jarring transitions as one moves from layer to layer. For example, a modeling package 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. To be able to add C or FORTRAN extensions to such software, one must first become familiar with many details of the software's C or FORTRAN *implementation.* Even worse, virtually none of the error checking and other safeguards provided at the higher level are available in C or FORTRAN. SLX users almost never find it necessary to drop down to a lower-level, more powerful language, because the SLX kernel language has an expressiveness approaching that of C. In addition, the SLX kernel language includes complete checking to prevent "shoot yourself in the foot" errors such as referencing beyond the end of an array and using invalid pointer variables, both of which are all too familiar to C programmers.

Reprint - Presented at the 1999 Winter Simulation Conference

C. SLX's mechanisms for moving from layer to layer are very powerful. These mechanisms are *abstraction* mechanisms. Higher levels provide more abstract descriptions than a lower levels; i.e., 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, which go well beyond C, 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.

D. SLX has excellent mechanisms for coupling SLX programs with other software. For example, if you have a collection of C functions you'd like to call from SLX, all you need to do is (1) place them into a Windows Dynamic Link Library (DLL), and (2) provide prototypes which tell SLX about the arguments and values returned by your functions. SLX can automatically generate C/C++ header files (.h files) which define SLX objects using C/C++ syntax. Thus the most error-prone step of establishing a cross-language interface, achieving exact agreement on the data structures used, has been automated. SLX's DLL interface is described in Section 4.1.

(Henriksen 1997) discusses how SLX was used to build a software package for modeling conveyor systems. (Brill and Whitney 1997) presents an example of the use of SLX for datailed traffic modeling. Both references provide examples of the exploitation of SLX's layered architecture.

In the sections which follow, SLX's extensibility mechanisms are illustrated; selected features of the SLX kernel are presented; and examples are presented which describe the coupling of SLX and other software. Finally, the ramifications of SLX on the teaching of simulation are discussed.

2 EXTENSIBILITY FEATURES

SLX is an extensible platform on which a wide variety of higher level simulation applications can be built. In this section we provide an overview of how the extensibility mechanisms work.

2.1 Unbounded, Executable Compiler Extensions

In a traditional language compiler, elements of a program (referred to below as *modules*) are translated into some form (referred to below as *object code*) which can be executed by a computer or interpreted by an interpreter program. The architecture of a traditional compiler is shown in Figure 2.

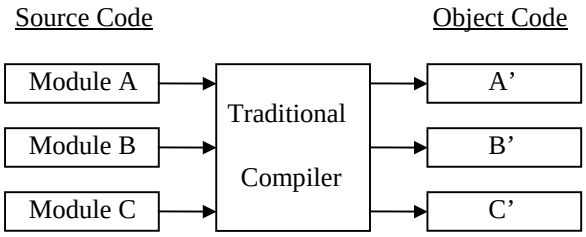

Figure 2: Traditional Compiler Architecture

In SLX, several source language constructs can be used to extend the SLX compiler. This architecture is shown in Figure 3.

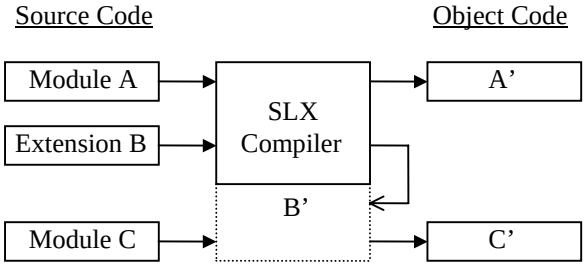

Figure 3: SLX Compiler Architecture

When the SLX compiler encounters the definition of a compiler extension, it sets aside its current work and processes the extension in its entirety. When the compiler resumes its work, the compiled extension is available for use throughout the rest of the compilation. In Figure 3, Module C can make use of extensions defined in Extension B. This process can be used repeatedly; i.e., the extended compiler can be further extended, without bound.

2.2 SLX's Statement Definition Facility

One of the most commonly used forms of SLX compiler extensions is the SLX statement definition facility. This facility allows the introduction of new *statements* into the SLX language. Such statements are similar to macros in traditional programming languages, except that they operate at the statement level, rather than at the expression level, as is commonly the case.

There are four major components of a statement definition:

A. a prototype which specifies the syntax of the statement (informally, "how it looks");
B. optional logic and looping within the definition, responding to the presence, absence, and other characteristics of statement components; and
C. one or more *expand* statements which inject "generated" text into the source stream seen by the SLX compiler.
D. optional *diagnose* statements which issue meaningful messages when errors in statement usage are made.

SLX statement prototypes are described using a meta-language which permits specification of the following kinds of statement components:

A. User-supplied expressions
B. User-defined keywords
C. Optional components
D. Repeated components; e.g., lists of items
E. Punctuation characters

Perhaps the most striking feature of all of SLX is the vehicle by which the logic, looping, expansion, and issuance of diagnostics are expressed. Most languages which have macros employ special sublanguages for defining macros. Typically such sublanguages are radically different from, and weaker in expressive power than, their host languages. For example, #if, #else, and #endif in the C language offer very weak capabilities for conditional expansion of macros, and their syntax differs from that of C itself. In SLX, there is no separate sublanguage used for statement definitions; rather, the SLX language itself is used. The only limitation is that simulation constructs such as time delays, fork, and wait until, which have no meaningful interpretation during program compilation, cannot be used.

The ability to use (almost) all of the SLX language in statement definitions permits tremendous flexibility and complexity in statement definitions. For example, a statement definition can read information from a file and store the information in user-defined, compile-time data structures which are interrogated and manipulated by other statement definitions.

In addition to statement definitions, SLX supports more traditional macros and *precursor* modules. Precursor modules are "large" SLX compiler extensions. They are not limited to just macros and statement definitions; rather, they can contain a host of functions and data which are to be made available at compile-time, run-time, or both. Finally, note that all three forms of

SLX compiler extensions (statement definitions, macros, and precursor modules) are compiled into executable machine instructions by SLX. Thus, SLX fulfills the promise of unbounded, executable user extension of SLX itself.

3 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 this section, we will present some representative features, to illustrate the functionality, ease-of-of-use, and *ease-of-learning* of SLX.

3.1 Objects and Pointers to Objects

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 a model of a factory, widgets being produced would be modeled as passive objects, since they have no self-determined, executable behavior. Their behavior results from being acted upon by other objects. (For those readers familiar with C, passive objects are very much like C structs.) *Active* objects have executable, at least partially self-determined behavior patterns. In a model of a factory, a foreman would be modeled as an active object.

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, a 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.

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 in Section 3.2) 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 butcher;
```

The manipulation of pucks is the basic mechanism by which a collection of objects experiences events over time. By rapidly switching from puck to puck, the SLX simulator creates the illusion of parallelism among the activities of the objects to which the pucks are attached. Scheduled time delays, e.g., service times, and state-based delays, e.g., waiting for a server to become available, are operations performed on pucks.

3.2 What's a Puck?

The original version of GPSS introduced the transaction-flow modeling world-view in 1962. In the transaction-flow world view, attention is focused on units of traffic, called transactions, which flow through the block diagram representation of a system, competing for system resources. In the 36-year period since GPSS was introduced, a large 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 (current block in the block diagram) of the unit of traffic in a model. Figure 4 illustrates this architecture. In a GPSS model of a supermarket, a transaction representing a shopper would have attributes such as probabilities of visiting various departments, e.g., the deli, expected number of items to be purchased in each department, etc. Scheduling data would include priority, next scheduled event 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.

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 instance of an SLX user-defined *object class*. The role of a transaction's scheduling data is played by an SLX *puck.* Each SLX object created is an instance of its object class

and has its own copy of the object class's data. 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. In SLX, it is possible to have more than one puck for a given object. An object instance for which there are two pucks is shown in Figure 5.

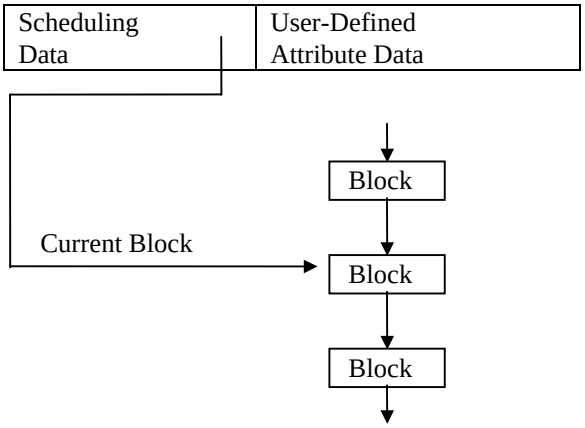

Figure 4: Traditional Transaction Architecture

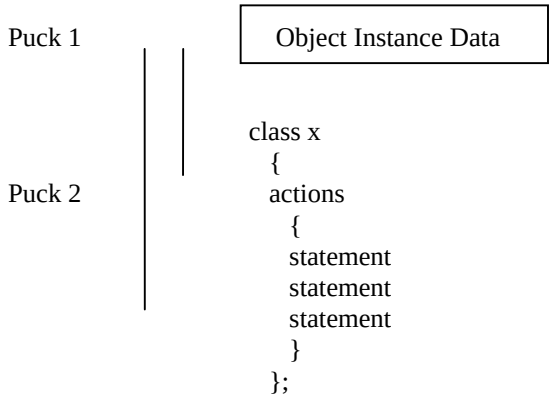

Figure 5: An Active Object With Two Pucks

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.) Inter-object parallelism, in which there is a 1:1 relationship between objects and pucks, is functionally equivalent to transaction flow. Intra-object parallelism is achieved by creating more than one puck for an active object. This is accomplished by means of a *fork* statement. Suppose that in developing a model of a factory, we need to model a complicated machine which is capable of performing three operations simultaneously. Some components of the machine are common to all three operations. The data describing such components must be easily accessible within the portions of the model for each of the three operations. Figure 6 shows how an active object can be used to model such a machine, using fork statements.

Each fork statement creates a new puck for the machine 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 next statement. After the second fork is executed, the machine object has three pucks, each of which has direct access to data common to the entire machine, and each of which is independently scheduled. Thus our active machine can do three things at once.

```
class machine
  {
  "Declarations for variables local to the machine"

  actions
    {
    fork
      {
      "actions for operation 1"
      }

    fork
      {
      "actions for operation 2"
      }

    "actions for operation 3"
    }
};
```

Figure 6: Intra-Object Parallelism Using Forks

Most transaction-flow simulation languages offer only inter-object parallelism. Most also 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). A new transaction is another complete instance of Figure 4. SLX's fork statement creates a new puck (scheduling data *only*) which shares the user-defined attributes with other pucks, as shown in Figure 5.

If a language has only a transaction-cloning verb, and no fork verb, modeling system components such as the

complicated machine discussed above is much more difficult, although certainly not impossible. Consider, for example, GPSS/H's SPLIT block, which creates a clone of an entire transaction. We could use SPLIT blocks to model our machine. The difficulty arises in choosing where to store the data that must be shared by all three transactions. 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 only one such machine exists, storing the shared data in global variables is easy. If there is more than one such machine, separate collections of shared global variables must be used, one collection for each such machine. If the collection of machines does not change during model execution, the shared data can be statically allocated. However, if the collection of machines 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 is an extremely handy modeling tool. In complex modeling situations, intra-object parallelism can be indispensable. 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.

3.4 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 a time-based condition.

```
optimistic_event_time = "some expression"

wait until      (time == optimistic_event_time
                 or "some other condtion");
```

4 SLX AS A COMPONENT OF YOUR WORLD

Although SLX is extremely powerful and flexible, there are situations in which it is convenient to use other software tools in conjunction with SLX. For example, if you have a pre-existing collection of C functions, it may be very handy to be able to call them from SLX. The remainder of this section provides examples of how SLX can be integrated with the other tools in your world.

4.1 SLX's DLL Interface

SLX has very powerful facilities for calling C/C++ functions which are contained in a DLL (dynamic link library). To call functions in a DLL, you must supply to SLX a function prototype which defines the arguments (if any) of each function, the values returned (if any), and the name of the DLL file. The SLX development environment has a menu item which can be clicked to generate a C/C++-compatible .h file which maps all SLX data passed to and from DLL functions into C syntax. SLX objects contain hidden elements which are used for error detection, debugging and other internal bookkeeping functions. If an SLX object is to be manipulated by a C function, the hidden information must be taken into account when constructing an analagous C/C++ struct definition. Accordingly, object elements for which there is a direct counterpart in C/C++ are described using straightforward declarations in a generated .h file, and hidden elements are declared

as arrays of bytes with the dimension chosen to "pad" the C/C++ struct to achieve agreement with SLX.

When SLX detects the first call of any function in a given DLL, it checks to see if the DLL has a function named "connect." If so, this function is called first, and SLX passes it a pointer to a vector of pointers to callback functions inside SLX. These functions can be used to perform functions that are risky or impossible to perform from C/C++ subsequently called DLL functions. At the completion of execution, each DLL used is interrogated for the existence of a "disconnect" function. Any such functions found are called by SLX prior to SLX program termination. This allows DLLs to perform any final "cleanup" operations, e.g., closing open files.

4.2 SLX-Proof Interface

Wolverine Software has developed an interface between SLX and Proof Animation (Henriksen 1998) using SLX's statement definition facility. Proof requires an input stream of ASCII commands that create and destroy objects on the screen, move them, change their colors, etc. A small, but powerful collection of commands is used for this purpose. SLX statements have been defined for generating the commonly used Proof commands and command options. The syntax of the SLX statements matches that of the corresponding Proof commands. For example, to generate a

```
place 27 on loop
```

Proof command, one might write

```
PA_place objectID on "loop";
```

In the example shown above, "27" and "loop" are variable components of the Proof *place on* command. The SLX code supplies "27" as the value of a variable named objectID and supplies "loop" as a string constant.

The current version of the SLX-Proof interface writes Proof command streams to files. A DLL version of Proof is under development. When this version is completed, the statement definitions in the SLX-Proof interface will be augmented to allow the transmission of commands directly to Proof.dll without using files. This will make it possible to run a simulation and animation concurrently.

A third party has developed an SLX package that is capable of reading entire Proof layout files, storing them in SLX data structures, and rewriting the layout files. Thus geometric characteristics of layouts drawn or modified using Proof are accessible to SLX programs. In addition, Proof layout files can be modified by an SLX program.

4.3 SLX-Prime Interface

Prime (Wagner and Wilson 1997) is a software package for fitting Bézier-curve-based probability distributions to data observations. Bézier curves can be fitted to data using a variety of automated algorithms and by *visual* manipulation of the control points which define the Bézier curve. Thus it is possible to take a fitted curve and move the mass of the probability distribution around. For example, one might feel that in a real system, data might be a little more skewed to the right than collected experimental data would suggest. Visual manipulation of the distribution makes this easy to do, provided that the resultant curve can be easily incorporated into a random variate generator in a simulation package.

The output of Prime is a collection of Bézier control points stored in a file in a straightforward ASCII format. In cooperation with the author of Prime, Wolverine Software developed several statement definitions which allow direct incorporation of Prime-generated curves into SLX models. The "Bézier_data" statement reads a Prime-generated file (at compile time!) and deposits the defined control points into an SLX object. This object can be subsequently used for generating random variates from the fitted distribution.

SLX and Prime work very well together, The initial integration of the two packages was accomplished in under 24 hours. After the initial integration, a highly tuned variate generator was written in assembly language, to achieve maximum efficiency in variate generation. This required another day's work.

4.4 SLX-HLA Interface

SLX's DLL interface has been used to connect SLX models with the run-time infrastructure (RTI) of HLA (DoD 1997), DoD's High Level Architecture for distributed simulations (Strassburger, Schulze, Klein, and Henriksen 1998). Integration was accomplished by building C++ wrapper functions which sit between SLX and the RTI. The integration of SLX and HLA is highly synergistic. It brings to SLX an architecture which promises to achieve widespread adoption for distributed, interoperable simulations. For people who know HLA and want to develop such simulations, SLX provides a powerful alternative to developing simulations from the ground up in a high-level language such as C++ or ADA.

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 (Schriber and Brunner 1997). 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 (time >= 5:00 and 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.

The hierarchical architecture of SLX is mirrored by Windows-based tools in the SLX model development / debugging environment. Windows can be opened to explore every aspect of puck management. Students of SLX have the ability to *see* how SLX works.

5 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. SLX has been used in a variety of very large, complex applications. Its extensibility mechanisms have been heavily exploited. SLX is easily integrated with other simulation tools, including HLA. If you're teaching or learning simulation, or developing simulations, SLX can be an invaluable component of your world. SLX stretches the boundaries of simulation software.

REFERENCES

Brill, J.C and D.E. Whitney. Development and Application of an Intermodal Mass Transit Simulation with Detailed Traffic Modeling. In *Proceedings of the 1997 Winter Simulation Conference*, ed. S Andradóttir, K.J. Healy, D.H. Withers, and B.L. Nelson. 1230-1235. Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

Crain, R.C. Simulation With GPSS/H. In *Proceedings of the 1998 Winter Simulation Conference*, ed. Madeiros, D.J., E. Watson, M.S. Manivannan, and J. Carson. Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

Department of Defense (DoD). High Level Architecture Interface Specification Version 1.2 (1997). Available on-line at http://hla.dmso.mil.

Henriksen, J.O., 1998 Windows-Based Animation with Proof. In *Proceedings of the 1998 Winter Simulation Conference*, ed. Madeiros, D.J., E. Watson, M.S. Manivannan, and J. Carson. Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

Henriksen, J.O., 1997 An Introduction to SLX. In *Proceedings of the 1997 Winter Simulation Conference*, ed. S Andradóttir, K.J. Healy, D.H. Withers, and B.L. Nelson. 559-566. Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

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.

Schriber, T.J. and D.T. Brunner. Inside Discrete-Event Simulation Software: How it Works and Why It Matters. In *Proceedings of the 1997 Winter Simulation Conference*, ed. S Andradóttir, K.J. Healy, D.H. Withers, and B.L. Nelson. 14-22. Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

Strassburger, S., T. Schulze, U. Klein, and J.O. Henriksen. 1998. Internet-Based Simulation Using Off-the-Shelf Simulation Tools and HLA. In *Proceedings of the 1998 Winter Simulation Conference*, ed. Madeiros, D.J., E. Watson, M.S. Manivannan, and J. Carson. Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

AUTHOR BIOGRAPHY

JAMES O. HENRIKSEN is the president of Wolverine Software Corporation. He was the chief developer of the first version of GPSS/H, of Proof Animation, and of SLX. He is a frequent contributor to the literature on simulation and has presented many papers at the Winter Simulation Conference. Mr. Henriksen has served as the Business Chair and General Chair of past Winter Simulation Conferences. He has also served on the Board of Directors of the conference as the ACM/SIGSIM representative.