Proceedings of the 1999 Winter Simulation Conference
P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans, eds.

SLX: PYRAMID POWER

James O. Henriksen

Wolverine Software Corporation
2111 Eisenhower Avenue, Suite 404
Alexandria, VA 22314-4679, U.S.A.

ABSTRACT

SLX is Wolverine Software's "next generation" simulation language. SLX stands for Simulation Language with Extensibility. SLX provides an inverted pyramid of layers which range 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. Building new layers atop old ones is facilitated by SLX's unique extensibility mechanisms. 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 examples of how SLX has been coupled with other software, respectively.

1 INTRODUCTION

SLX is a unique simulation language:

- It is implemented as a multiplicity of layers, rather than a collection of monolithic, "black box" building blocks. In SLX, you can get inside higher-level building blocks by "drilling down" to lower layers.
- SLX has extensibility, or layering mechanisms which allow you to expand existing layers or build new ones atop old ones.
- SLX has provides a development environment which allows you to put a model under the microscope through the use of windows which mirror a model's hierarchy of layers.

The layers of SLX form an inverted pyramid, as shown in Figure 1. Traditional language-based simulation tools fall in the middle of the SLX pyramidal hierarchy. By "traditional" we mean simulation tools that provide a set of building blocks which are combined and connected to form a network or block diagram representation of a system being simulated. This approach to model building has been used for a long time, at least since the advent of GPSS in 1962, if not earlier.

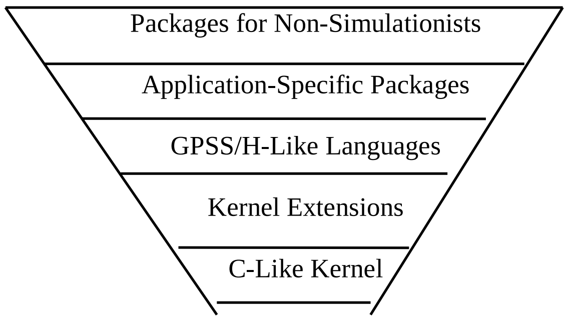
Packages for Non-Simulationists
Application-Specific Packages
GPSS/H-Like Languages
Kernel Extensions
C-Like Kernel

Figure 1: The SLX Pyramid

In the sections which follow, we will examine the strengths and weaknesses of traditional simulation software and explain how SLX expands upon the capabilities of such tools.

1.1 Strengths and Weaknesses of Traditional Simulation Software

The traditional, building-block approach to modeling offers a number of advantages. A well-conceived collection of building blocks is easily mastered, applicable to a wide variety of systems, and provides a good framework for conceptualizing models.

There are also a few problems with the traditional approach. First, the collection of building blocks tends to grow over time, as the tool's vendor extends and enhances the product. No matter how well this is done, some additions are obviously "grafted onto" the product, detracting from the conceptual integrity of the original collection. Second, no matter how well-conceived the collection of building blocks is, it's possible to encounter new modeling situations in which none of the available building blocks exactly fits a particular need. Third, for some users of simulation, a level of abstraction higher than that of the basic building blocks is appropriate. For example, a foreman on a factory floor might be able to make excellent use of a simulation, but be unable to write one. For him/her, a higher level of abstraction is necessary in order to specify model inputs and parameters, and to evaluate model outputs.

1.2 Improving on the Traditional Approach

SLX is the result of Wolverine Software's many years of experience in developing traditional, building-block languages. In the development of SLX, we were concerned with three primary issues: layers, layering mechanisms, and the model development environment which would provide access to the layers. Our first objective was to devise a hierarchy of well-conceived layers such that users of SLX would be able to spend most of their time using higher layers of the software, but be able to "drill down" into lower layers when appropriate. Our second objective was to develop layering mechanisms which made moving up and down the hierarchy easy. Our third objective was to provide Windows-based model development tools which were (1) hidden at higher levels of abstraction, (2) instantly available on demand, and (3) able to provide insight into the innermost (lowest level) workings of SLX

1.3 The Layers of the SLX Pyramid

The pyramidal appearance of Figure 1 depicts the size and diversity of SLX layers. The SLX kernel is a small, but very powerful collection of simulation primitives. It is the result of carefully studying our own GPSS/H language and GPSS/H-like building blocks to identify their common, "root" functionality. SLX's kernel has the following distinguishing characteristics:

- The number of required simulation primitives is astonishingly small (about a dozen, depending on how you choose to count them).
- Due to their atomic nature, kernel primitives can be combined in many ways, providing a foundation which supports a wide variety of higher level modeling paradigms.
- Kernel primitives are directly accessible. Traditional simulation software may include a number of building blocks which share a common underlying feature, but provide no direct access to the required feature, *per se*. For example, many building blocks require that traffic flowing through them be forced to wait until one or more components of the system attain particular states. Different building blocks may exist for acquiring control of a resource, waiting for a switch to become true or false, waiting for the contents of a queue to exceed a threshold value, etc. Each of these requirements can be fulfilled by SLX's kernel-level *wait until* statement.
- Access to kernel level primitives guarantees that there are no language-imposed limitations on model fidelity. The phrase "precision modeling" has come into vogue for describing very detailed simulations. In the SLX pyramid, the lower you go, the more precise you can be. At the kernel level, precision is virtually unlimited.
- The kernel provides C-like computational capability. Most traditional simulation software is lacking in computational capabilities, perhaps because greater emphasis is placed on features for managing parallel activities over time, than on computation.
- All kernel primitives, including those modeled after the C language, are implemented with exhaustive run-time error checking. Errors such as storing off the end of an array or using an invalid pointer value (the bane of a C programmer's existence) are trapped and diagnosed during model execution. Thus as one moves down the hierarchy, protection against errors is always present, even at the lowest levels.

Each of the higher layers in Figure 1 is wider than the layer below it. This is because higher layers include combinations of lower-layer features, with each combination tailored to fulfilling a particular requirement. Consider the SEIZE block, a GPSS/H building block used to acquire a single server in queueing systems. In SLX, the SEIZE block is implemented as a subroutine comprising only 9 lines of executable kernel-level code. The most important of these 9 lines of code are a wait until, waiting for the single server to be idle, and code for invoking statistics collection features contained in the layer between GPSS/H and the SLX kernel. The 9 lines are remarkable for two reasons: first, a 9-line subroutine is astonishingly small; and second, those 9 lines can be directly examined or stepped through by SLX users. What would be a black box in other tools is directly accessible in SLX.

1.4 Layering (Extensibility) Mechanisms

SLX's layers are only a part of the SLX story. Of equal importance are SLX's layering mechanisms, which make it easy to extend existing layers or build new layers atop old ones. Higher layers provide more abstract descriptions than a lower layers; i.e., lower-level implementation details are hidden at the upper layers. 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.

1.5 The SLX Model Development Environment

The final portion of the SLX story is SLX's model development and debugging environment. The SLX debugger provides unprecedented features for "putting a simulation model under the microscope." A variety of Windows can be opened during model execution to show model data, model source code, and simulation data structures such as event lists. Source code displays allow you to watch, at a desired level of detail, code as it is executed. Consider, for example, the GPSS/H SEIZE block discussed above. If one were stepping through a GPSS/H model, one could choose either to treat the SEIZE block as an atomic statement, or to step into its lower-level implementation. Figure 2 shows SLX's Calls & Expansions window, which depicts where you are in a model and how you got there. Figure 2 shows the state of affairs when a user has stepped all the way into the statistics collection code supporting the SEIZE block. By clicking on various levels of the Calls & Expansions tree, one can quickly move from layer to layer. This allows examination of implementation details when necessary and quickly determining "how on earth did I get to this point in my model."

Data windows can be opened for global data and data unique to individual units of traffic flowing through a model. Such windows are updated as data values change.

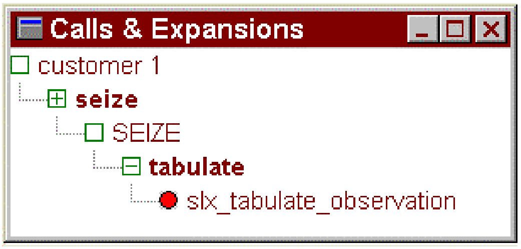

Figure 2: SLX Calls & Expansions Window

A wide variety of information is available by "right clicking" on data and/or source code displayed in SLX windows. Figure 3 shows sample options for viewing data, and Figure 4 shows options for viewing source code and/or setting debugger breakpoints.

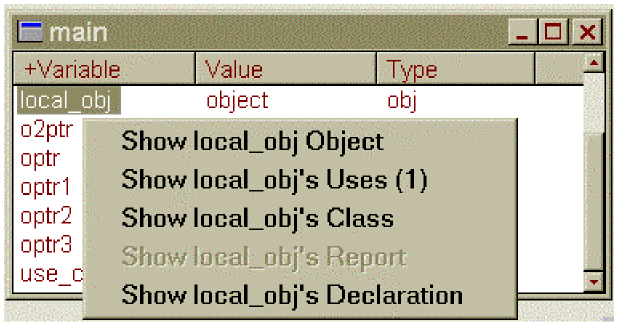

Figure 3: Sample Data Display Options

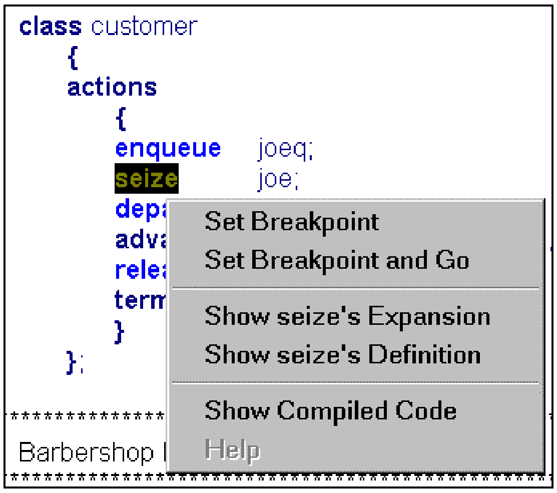
```
class customer
    {
    actions
        {
        enqueue    joeq;
        seize      joe;
        depa
        adva
        rele
        term
        }
    };
```

Figure 4: Sample Code Display/Breakpoint Options

Collectively, SLX's layers, layering mechanisms, and model development/debugging environment comprise a unique, powerful simulation tool.

In the sections which follow, SLX's extensibility (layering) mechanisms are illustrated; selected features of the SLX kernel are presented; the ability to integrate SLX with other software is described; and examples are cited which illustrate how SLX has been used in a variety of large-scale projects.

2 EXTENSIBILITY FEATURES

The SLX pyramid supports development of a wide variety of higher level simulation applications. Much of the power of SLX derives from the ease with which one can extend existing layers or build new ones atop old ones. In this section we provide an overview of how SLX's extensibility mechanisms facilitate such efforts.

2.1 SLX Compiler Extensions

Extensions to the SLX compiler are available in three forms:

- SLX macros provide the simplest form of extension. SLX macros are similar to macros found in many programming languages, spreadsheets, etc.
- SLX statement definitions allow the introduction of new *statements* into SLX. We have used statement definitions to provide SLX equivalents to many GPSS/H blocks.
- SLX precursor modules provide a mechanism for grouping together collections of compiler extensions into packages which can incorporate very powerful compile-time functionality.

2.2 Unbounded 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 5.

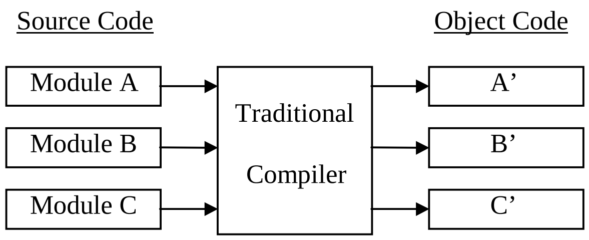

Figure 5: Traditional Compiler Architecture

In SLX, macros, statement definitions, and precursor modules can be used to extend the SLX compiler. This architecture is shown in Figure 6.

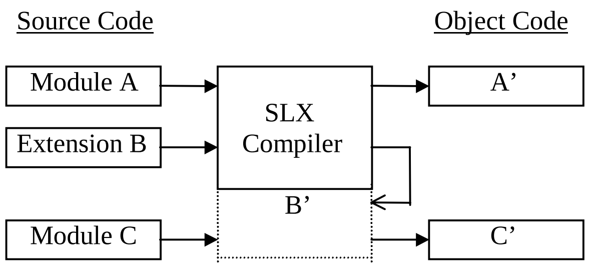

Figure 6: SLX Compiler Architecture

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 6, 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.

2.3 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, "what it looks like");
B. optional logic and looping within the definition, responding to the presence, absence, and other characteristics of statement components;
C. one or more *expand* statements which inject "generated" text into the source stream seen by the SLX compiler; and
D. optional *diagnose* statements which issue mean-ingful 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 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, no separate sublanguage is used; 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.

As stated at the end of Section 2.2, extensions can be built upon extensions upon extensions, without bound. One might ask whether the complexity of model translation becomes unwieldy in such circumstances. All three forms of SLX compiler extensions (statement definitions, macros, and precursor modules) are compiled into executable machine instructions by SLX. Thus, there is no translation performance penalty for heavy use of extensions. The process of cumulative extension of SLX is therefore described as “unbounded, executable, end-user extension.”

3 SLX KERNEL FEATURES

The number of primitives required to support simulation is surprisingly small. However, for a simulation software developer, implementing some of these primitives in a general form can be quite 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-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 transac-tion-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 37-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, (1) user-defined data particular to the unit of traffic, and (2) “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 7 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.

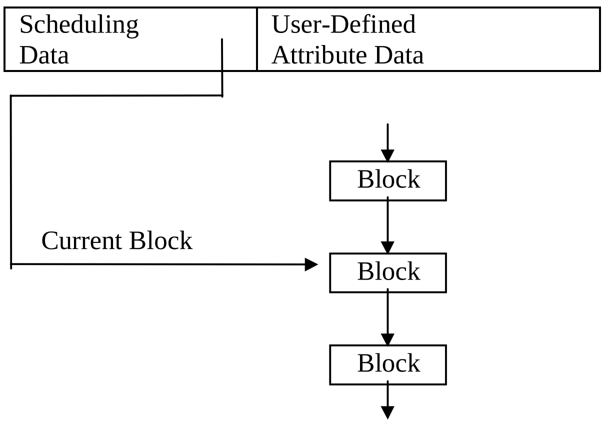

Figure 7: Traditional Transaction Architecture

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 8.

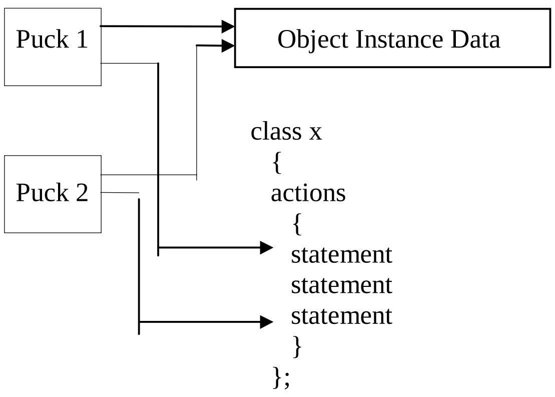

Figure 8: An Active Object With Two Pucks

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 9 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 9: 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 7. SLX's fork statement creates a new puck (scheduling data *only*) which shares the user-defined attributes with other pucks, as shown in Figure 8.

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 condition");
```

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 1999) 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 SLX-Proof interface can either write Proof command streams to files for post-processed animation or transmit them directly to the DLL version of Proof for concurrent or even real-time animation.

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-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 APPLICATIONS OF SLX

SLX has been used in a variety of complex, large-scale applications:

- Large, multi-modal transportation center
- Material-handling & Conveyor Systems
- HLA federation component
- HLA animator (with Proof Animation)
- Telecommunication systems
- Pneumatic tube hospital specimen delivery system

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 = pyramid power.

REFERENCES

Brill, J.C and D.E. Whitney. 1997. 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. 1998. *Proceedings of the 1998 Winter Simulation Conference*, ed. Madeiros, D.J., E. Watson, M.S. Manivannan, and J. Carson. 235-240. 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., 1999 General-Purpose Concurrent and Post-Processed Animation with Proof. In *Proceedings of the 1999 Winter Simulation Conference*, ed. P.A. Farrington, H.B. Nembhard, G.W. Evans, and D. Sturrock. 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. 1997. 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. 1669-1676. 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.