SLX

Frequently Asked Questions

What is SLX?

SLX is a Windows-based system for developing discrete event simulations. Its advanced technology accommodates a wide range of usage styles. If you're a beginning simulationist, you can use prepackaged features to do your work. If you're a simulation expert, you can use SLX to build simulation packages tailored to specific applications for use by others. If you're neither a beginner nor an expert, you may want to pick and choose among built-in features, tools provided by others, and tools you create yourself.

SLX is a compact system, so you don't need to learn hundreds of features to use it. The handful of features comprising the core of SLX have been carefully distilled from our many years of experience developing simulation software. SLX has a very open architecture, with no "black boxes". If you need to know what's going on in a model, you can use SLX's browser and visually-oriented tools to place not only the model, but also SLX itself, under the microscope.

What does SLX stand for?

SLX stands for Simulation Language with eXtensibility.

What do you mean by extensibility, and why is it important?

Extensibility mechanisms allow you to build new features out of old ones. While any language which has objects, macros, and subroutines possesses a degree of extensibility, the mechanisms in SLX go well beyond those found in other languages. For example, SLX has a statement-definition facility with which you can add new *statements* to the language. This enables you to develop a dialect of SLX which is custom-tailored for your application. For example, if your models frequently deal with conveyors, writing them using verbs such as "place on" and "remove from" is much easier than using general-purpose simulation verbs.

Why should I use a *language* in a point-and-click world?

There's no question that graphically-based modeling systems are very popular. If you're trying to complete a modeling project in 3 days, the first-day ease-of-use afforded by such packages may be attractive. But if you're doing a more extensive modeling project, sooner or later you'll reach a point at which you must provide a detailed procedural description of how one or more components of your system really works. For example, your system may be computer-controlled and have the ability to react to a variety of inputs in complex ways. Describing complicated procedural behavior is hard to do using purely graphical tools – a bit like kissing through a screen door. In contrast, SLX's descriptive capabilities are virtually unlimited. When you know what you want to say, typing a few lines of code can be much easier than fighting with a poorly designed graphical system which is ill-suited for the task at hand.

In addition, characterizing SLX as simply a *language* can be very misleading. If your concept of what constitutes a language is based on experience with traditional simulation or programming languages, you'll be amazed at the difference.

Why is it important to have multiple layers in a simulation package?

Single-layer packages can trap you in the "90% syndrome". When you use such a package, which initially appears to be well-suited for your application, you can make rapid progress to the point at which your model is 90% complete. However, as you attempt to add final details, you may find that none of the built-in tools exactly meets your requirements. A menu which offers 20 options is useless if what you need is the 21st and you can't easily add a new option. With a multi-layer architecture, if a given layer lacks what you need, you can drop down to a lower layer and build what you need using lower-level tools. You can't get painted into a corner.

Why are the layers in SLX so good?

1. The layers are well conceived, as a result of Wolverine's over twenty years' experience building simulation software.
2. The layers are not too far apart. Many simulation tools allow you to drop down to the level of writing C or Fortran code, but this layer is too far removed from the normal modeling layer. To operate at this level, you first have to become familiar with a package's implementation details – often a tall order. In contrast, SLX provides a larger number of layers which are closer together, so you can shift easily from layer to layer instead of taking a jarring plunge into the world of C or Fortran.
3. The kernel, or bottom layer of SLX, provides access to "rock bottom" simulation primitives. We have yet to encounter a situation in which a necessary modeling feature cannot be built on top of SLX's kernel.
4. SLX's extensibility mechanisms make it easy to build higher layers on top of lower layers, or to add another tool within a layer.
5. SLX's development environment provides visually-oriented tools for exploring the layers of a model. For example, SLX's "Calls & Expansions" window displays a tree view of your model, providing single-click access to source code for all currently active layers. If a modeling error occurs at a low level within SLX, you can click higher in the tree view to see how your model got to the point of the error. If you want to step through a model in great detail, you can step over, step into, or step out of procedure calls or defined statements.

How easy is SLX to learn?

As with most things in life, it depends. We recommend learning SLX from the bottom up. There are only 8-10 lowest-level simulation primitives (depending on what you think counts as a simulation primitive). Simple models can be constructed using only four of these primitives. In the document entitled "Simulation Using SLX," we introduce SLX modeling in a sequence of eight small exercises of increasing complexity. These eight exercises cover the most important simulation primitives. Once you have mastered this small collection of primitives plus a few others, you can simulate virtually anything. As you move on to higher level language constructs, you will already have an understanding of their underpinnings. This approach to teaching SLX differs from traditional teaching methods. The usual approach to teaching/learning simulation is to start out with higher-level constructs, develop familiarity with their operation, and only at the end – if ever – explore how they really work. The bottom-up approach we recommend for learning SLX allows you to simultaneously learn what discrete-event simulation is all about and what SLX language constructs are needed to accomplish the job.

How well does SLX handle errors?

First, SLX provides clear, concise descriptions of errors, which are highlighted in red at their point of occurrence. Second, models run in a *totally* secure execution environment. Errors such as accessing beyond the end of an array, referring to NULL pointers, and dividing by zero are all detected and highlighted in red at their point of occurrence. By their very nature, simulations exhibit random behavior. If not detected by the system, errors such as accessing outside the bounds of an array could go unnoticed yet cause illegitimate "random" behavior. SLX has a "zero tolerance" approach to all such errors.

What features does SLX have which are truly unique?

SLX contains many innovations. The three most important are its capabilities for expressing parallelism, its "wait until" mechanism, and its compiled macros and statement definitions.

1. The most important reason one uses a simulation language is to be able to describe processes that take place in parallel. The ease with which this can be accomplished is an important measure of the power of any simulation tool. SLX provides very powerful mechanisms for describing parallel processes. In SLX, models are built by describing a system as a collection of *objects*. For each *class* of objects, attributes are defined which describe the class's characteristics. For example, in a model of a harbor, an object class representing ships might have attributes of tonnage, departure time, and cargo type. At any given time, a system can contain multiple *instances* of a given object class. For example, in a model of a harbor there will be many instances of ship objects, each with its own unique tonnage, departure time, and cargo type.

   Some objects are only acted upon by other objects. In SLX, such objects are called *passive* objects. For example, in an airport model a runway would be represented as a passive object, since it cannot do anything on its own.

   Objects that can do things on their own are called *active* objects. The behavior of an active object is described in its *actions* property, which is a sequence of executable statements located in the object's class definition. SLX uses *pucks* to manage the behavior of active objects. A puck identifies the statement currently being executed in the object's actions property. The name puck was chosen because one can envision the statements comprising an actions property as being written on a sheet of ice, and the progress of an object through its actions property as movement of its puck across the sheet of ice. Each active object has its own puck, so at any given time many pucks can be moving through the actions property of an object class. While active objects are operating in parallel, they may also interact with one another. For example, in a supermarket model, customer objects must compete for the services of a butcher. The ability to easily describe inter-object parallelism of this type is commonplace in simulation languages.

   SLX goes beyond inter-object parallelism, however, to support intra-object parallelism as well. SLX has a primitive, *fork*, for describing parallelism within a given object. When an object executes a fork statement, a new puck is created for that object. All of a given object's pucks – no matter how many – share that object's attribute values. Intra-object parallelism is very handy for describing the operation of complex objects that can do more than one thing at the same time. For example, a machine may simultaneously machine a part, load a new part, and dispose of a previous part. Through the use of the fork statement, these three processes can be described within a single actions property. While one could define separate object classes for these three

processes, if a machine object has many attributes, deciding which attributes should be placed in which class definition could be difficult. It might even be necessary to define a fourth class just to act as a repository of attributes shared by the other three classes. SLX is the *only* puck-based simulation language. In other languages, inter-object parallelism is the only form of parallelism which can be described.

2. The second most important SLX innovation is its generalized "wait until" capability. This capability allows model components to wait for another component to attain a specified state, or for a collection of components to simultaneously attain specified states. Consider the grandfather of all queuing models, the barbershop. When a student constructs his or her first barbershop model, how does the barbershop shut down at the end of the day? In a first model, the shop probably closes at 5:00, ignoring a haircut in progress, if any, and ignoring customers, if any, waiting for the barber. In a second model, more realistic shutdown conditions would be used. The shutdown rule should be "wait until it's 5:00 or later; shut the door; and wait until the queue is empty and the barber is idle." In SLX, such conditions are easily described with wait until statements.
3. The third most important SLX innovation is its macro- and statement-definition facility. Many computer languages have the capability for defining macros. To define a macro, one first defines a prototype of the macro (what it "looks like"), and then one defines rules by which the shorthand notation in the macro is expanded into lower-level language elements. In most computer languages macro expansion rules are described using special macro "sub-languages" which are different from, and much less powerful than, the host language. Consider, for example, #if, #else, and #endif in the macro sub-language of C/C++. These primitive constructs are far weaker than the full range of conditional branching and looping statements in the C/C++ language itself. In SLX, however, the macro sub-language is not a *sub*-language at all – it's full SLX!

   When the SLX compiler encounters a macro- or statement-definition, it sets aside what it's currently doing and compiles the definition all the way into executable form. The compiled definition is then *immediately* available for use in compiling the rest of the program. Although SLX macros and statement definitions are user-provided, they become *executable* extensions of the SLX compiler itself. Since the full range of SLX statements can be used in the definitions of macros and statements, arbitrarily complex logic can be specified. For example, one could define a statement whose expansion reads a file and builds compile-time data structures reflecting the contents of the file. Other statement definitions could access the compile-time data structures and use the collected information to generate tailored sequences of lower-level SLX statements. This architecture provides *unbounded* extensibility!

What kinds of support tools come with SLX?

SLX comes with a development environment that includes an integrated editor, debugger, and browser. SLX makes heavy use of Rich Text Format (RTF) files. SLX's editor is RTF-based, and SLX's output is written in RTF format. This allows the use of proportional fonts for both model source and model output. Both source and output can be cut and pasted by RTF-aware word-processing software, e.g., Microsoft Word. For production runs of simulation models, SLX can be executed outside of the development environment.

What kinds of statistical tools does SLX have?

SLX has an excellent collection of tools for generating random variates, collecting observations of random variables, and analyzing random outputs. Variate generators are included for 39 named distributions, for discrete and continuous empirical distributions,

and for Bezier curve-fitted distributions. SLX has innovative methods of collecting observations of random variables. For example, one can define a random variable to be observed and specify that it is to be collected over a number of formally defined (named) collection intervals. These intervals can be started, stopped, and restarted under model control. Thus, it is easy to collect hourly, daily, and weekly statistics for a random variable. Warmup and steady-state statistics can be collected using the same mechanisms. Finally, SLX has built-in procedures for building confidence intervals, analyzing correlation, and producing summary reports, including histograms.

Is SLX object-oriented?

SLX1 was object based, but not object-oriented. SLX2 introduced object-oriented features to SLX, borrowing ideas from C++ and Java, but paring down their complexity and packaging them in ways that help prevent users from shooting themselves in their feet. SLX3 added improved options for supporting both object-oriented approaches and more limited SLX1 approaches. Most object-oriented tools achieve extensibility through defining class hierarchies. While this approach is possible in SLX, SLX's powerful macro and statement definition facilities add another dimension to extensibility.

Why shouldn't I develop simulations in C++?

If you don't already know C++, you'll find that it's a large, complex language which is *much* harder to learn than SLX.

There's no question that over the broad range of software applications, C++ is much more powerful than SLX. When one is concerned specifically with developing simulations, however, broad-based power is less important than suitability for the task at hand. C++ is deficient in several key respects for use in developing discrete event simulations.

In C++, it is easy to make undetected errors that have devastating consequences. Anyone who has ever programmed with pointers knows the problems that occur when invalid pointer references are made. In SLX, by contrast, all pointer references are validated at run-time.

C++ contains no native capabilities for parallelism. C++ has a stack-based architecture, so in order to suspend and resume parts of a C++ program that correspond to simulation processes, one must either use operating-system based multi-threading (with enormous loss of efficiency) or write assembly language routines which save the state of the stack on suspension and restore it on resumption. The latter approach is inefficient, and it imposes modeling limitations as well. (Local data can be accessed only for "live" processes.) SLX's powerful capabilities for describing parallelism simply have no counterpart in C++.

One cannot simply add a few object classes to C++ and do meaningful simulation. To do simulation, one needs a simulation executive program, an event list algorithm, a library of random variate generators (SLX has 39), and routines for collecting and analyzing statistics. The development of such tools cannot be done in days or weeks; it requires months and years. If you don't believe this, you haven't developed such software.

The SLX debugger has been specially designed to be aware of simulation-specific actions and data items. A C++ debugger has no knowledge of such things, so setting breakpoints and monitoring a model's state would be extremely difficult.

Finally, one must consider the mechanics of editing, compiling, linking, and executing C++ code. SLX compiles a model directly into memory for immediate execution. Models are compiled by SLX at a rate of several hundred thousand lines per second. If you correct an

error in a 5,000-line SLX model, you can be conducting your next test in a fraction of a second. How long do you think it takes a typical C++ system to compile and link a 5,000-line program?

Can SLX handle really large models?

SLX is capable of handling *extremely* large models. Compiling a 10,000-line model is effortless. Moreover, the SLX compiler generates native machine instructions, yielding very fast execution. SLX includes highly optimized algorithms for event management and other critical simulation tasks. For example, SLX's event list algorithm was developed and perfected for GPSS/H, and its superiority has been demonstrated over a 20-year period. SLX is *very* fast.

How does SLX relate to GPSS/H?

GPSS/H is a very powerful simulation tool. It has a distinguished 30-year track record, and it has been used for a vast array of applications. We anticipate that it will be used for many more.

While SLX is not a replacement for GPSS/H, it contains a lot of its spirit. For example, SLX's "zero tolerance" policy for errors was derived from a similar policy in GPSS/H. In SLX we tried to build on the strengths of GPSS/H, eliminate cumbersome or rarely used features, combine related features into more general features, and package everything in an improved modeling environment. If you know and love GPSS/H, with a bit of gear-shifting, you can transfer your knowledge to SLX. We have implemented a subset of GPSS/H in SLX, so the familiar SEIZE, RELEASE, ENTER, LEAVE, QUEUE, DEPART, ADVANCE, and TERMINATE (among others) are available in SLX. GPSS/H's GENERATE has been recast as SLX's "arrivals" statement.

Does SLX have animation?

SLX works hand-in-glove with Wolverine's Proof Animation software, P5 (2D) and P3D (3D). Proof is driven by a stream of *trace* commands. For post-processed animation, the command stream is supplied in the form of a trace file. For concurrent animation (with a simulation and animation running at the same time), the trace stream is supplied by calling animation library (DLL) functions. Virtually all new PCs have at least two CPU cores, enabling a simulation to run in one core and an animation to run in the other. SLX includes an interface package that defines SLX statements for easily generating trace streams in SLX models. For each Proof trace command, there's a corresponding SLX statement with a 1-to-1 relationship in syntax.

Can I use C/C++ code I already have in conjunction with SLX?

Yes. Programs written in other languages can be used with SLX by placing them into standard Win32 DLLs (dynamic link libraries). SLX's "File" menu has an option for generating C/C++ header (".h") files containing prototypes for DLL routines that will be called from SLX and definitions for the data structures that are passed to them. We have automated the hard part of the cross-language interface, making sure that everyone agrees on the format of data exchanged by the languages involved.