Chapter One

INTRODUCTION

1.1 Introduction

A discrete-event system is one in which all state changes (events) take place in, or without loss of accuracy, can be viewed to take place instantaneously. For example, in a discrete-event simulation of a supermarket, customers arrive at the front door at precise instants in time. Systems in which state changes take place over nonzero intervals of time are known as *continuous* systems. If we were concerned with keeping a supermarket warm in the winter, we might simulate complex airflows over time, using differential equations. In discrete-event simulations, time is viewed as a progression of *instants*. At each instant in time, all events that can take place at that instant are processed, and when all such events have been processed, the simulator clock is advanced to the next imminent event time. Note that in some special-purpose simulations, a fixed-increment clock may be used, but for most simulations, this approach would result in many instants at which no events took place, and would therefore be very inefficient.

Components of discrete-event systems are either active or passive. The behavior of *active* components is determined by executing user-specified logic. For example, in a model of a supermarket, customers would be active components, each following a set of rules that contain randomness. A given customer might have a 5% probability of needing service at the deli counter. *Passive* components are acted upon by active components and have no "executable" behavior patterns. In a supermarket, a parking lot would be a passive component. It would be represented as a data structure that reflects availability of individual parking spaces, traffic flow, etc. Whether a system component can be represented as a passive object often depends on the level of detail. In a simple supermarket model, a butcher might be represented as a passive server who simply responds to customer demands. In a more detailed model, the butcher would have to occasionally take time to rearrange shelves, grind meat, etc. Describing such rules would require modeling the butcher as an active component. Finally, note that some components may alternate between active and passive roles. A parking garage attendant would have a very active behavior pattern most of the time, but if he needed to take an elevator to get back to the entrance of the parking garage, he might temporarily assume a passive role.

In a discrete-event system, components undergo two kinds of delays. *Scheduled* delays are delays for which a completion time can be computed. For example, if a server is uninterruptible, the completion time for performing a given service can be scheduled. The other type of delay is the *state-based* delay. Examples of state-based delays include "Wait for the light to turn green," or "Don't shoot 'til you see the whites of their eyes." Frequently, delays occur in which both a scheduled completion time and a state-based condition apply, but we don't know which will occur first. For example, if we go to the bank on our lunch hour, if we have to wait in line more than five minutes, we may give up (renege). When a combined scheduled/state-based delay is initiated, the likeliest outcome depends on the nature of the delay. In our bank example, we expect that getting served (state-based) is the normal case, and reneging (time-based) is the exceptional case. Conversely, if we're modeling a conveyor belt and place an object on the conveyor, and we compute the time to get from point A to point B, we expect that most of the time, things will go as predicted, but occasionally, a state-based condition may take precedence. For example, we might have to stop the belt, due to a blockage elsewhere.

1.2 SLX Innovations

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.

1.2.1 Representing Parallelism

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's components as *objects*. For each *class* of objects, attributes are defined that 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.

In SLX, 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. An active object can experience both scheduled and state-based delays. In most systems, active objects compete for the use of passive objects. Since many objects can be active at any given time, inter-object parallelism is commonplace

SLX goes beyond inter-object parallelism, however, to support *intra*-object parallelism as well. In SLX, it is extremely easy to schedule multiple concurrent activities for an active object. Consider a complex machine that can simultaneously load a new part, machine a part, and dispose of a completed part. Most simulation languages offer only inter-object parallelism. In the absence of intra-object parallelism, we would have to represent the machine described above as three active objects. If the representation of the machine requires a complex data structure, one of the big difficulties is deciding in which of the three objects various data should be placed, and how access to the data is shared among the three active objects.

1.2.2 SLX's Wait Until

The second most important SLX innovation is its generalized "wait until" capability. This capability allows active objects to wait for a combination of state-based and time-based conditions to be attained. 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 simple model, the shop might close at 5:00, ignoring a haircut in progress, if any, and ignoring customers, if any, waiting for the barber. In a more realistic model, shutdown conditions would be both time- and state-based, e.g., "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, it's very easy to describe such conditions by using the "wait until" statement.

1.2.3 SLX's Extensibility Mechanisms

The third most important SLX innovation is its macro- and statement-definition facility. SLX's defined statements can be regarded as "super macros" that are used at the statement level in SLX programs. 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" that 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, 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 (machine instructions). 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!

1.3 The Architecture of SLX

SLX has a layered, inverted pyramidal architecture, as shown in Figure 1.

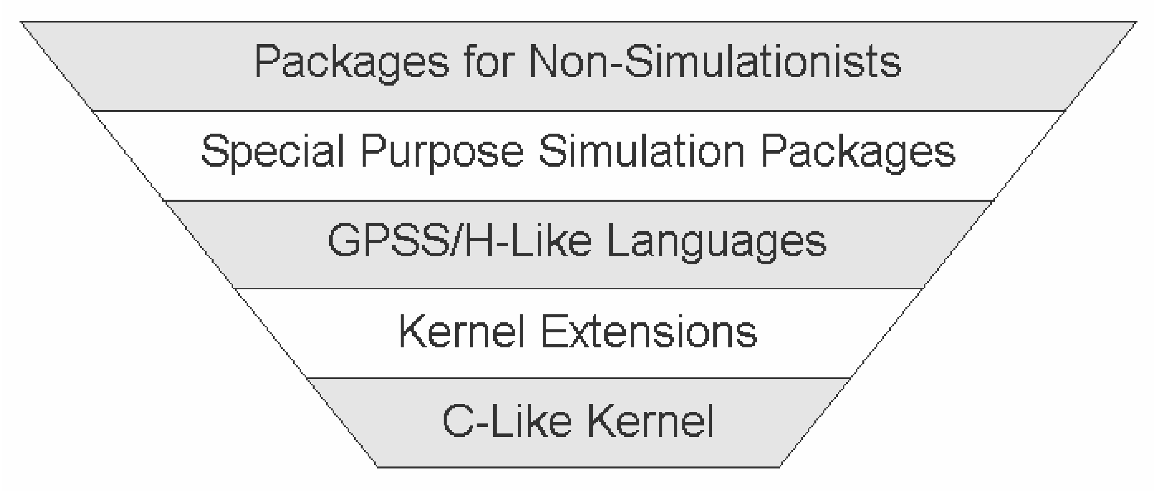
Packages for Non-Simulationists
Special Purpose Simulation Packages
GPSS/H-Like Languages
Kernel Extensions
C-Like Kernel

Figure 1: The SLX Pyramid

Traditional language-based simulation tools fall in the middle of the SLX pyramidal hierarchy. As is evident in Figure 1, SLX's layers extend both below and above the focus of traditional simulation languages.

The bottom layer of SLX is a C-like kernel language, with a few features influenced by, but not necessarily copied verbatim from C++. Constructs of C which are error-prone or intended primarily for systems programmers were excluded from or restricted in the SLX kernel. Conversely, discrete event simulation primitives such as simulated time, parallelism, scheduling, and synchronization, not found in C, were added to the SLX kernel, using C-like syntax. SLX's kernel is a small, but very powerful language for constructing simulations at a "nuts-and-bolts" level. It provides the underlying support for higher levels in the pyramid.

The top layers of SLX extend above the focus of traditional simulation languages. SLX's extensibility mechanisms can be used to develop dialects of SLX in which nouns and verbs are application-specific.

The efficacy of SLX's layered approach is hinges on four key factors:

1.3.1 The Contents of SLX's Layers

SLX's layers are well-conceived. In each layer, we have taken a minimalist-generalist approach, providing only those capabilities that are absolutely necessary, but implementing them in as general a manner as possible. Consider the design of SLX's kernel. We went to lengths to minimize the "footprint" of the kernel. As a result of our approach, the SLX kernel is a surprisingly small collection of precisely defined, very general primitives that can support a wide variety of higher-level modeling approaches. SLX's kernel-level *wait until*, which we discussed in Section 1.2.2, is a good example. This single statement provides as a foundation for all state-based events. Although higher layers may implement a variety of world views, e.g., transaction flow, process interaction, activity scan, and Petri nets, all of these world views require state-based events.

1.3.2 Separation of SLX Layers

SLX's layers are properly separated. Many modeling tools provide multiple layers, but often these tools exhibit wide gulfs between their 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 Visual Basic. The problem with this approach is that there are only two layers, and they're too far apart. To be able to add C or Visual Basic extensions to such software, one must first become familiar with many details of the software's implementation. Even worse, virtually none of the error checking and other safeguards provided at the higher level are available in C or Visual Basic. 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.

1.3.3 Building New Layers on Top of Old Layers

SLX provides very powerful *abstraction* mechanisms for moving from layer to layer. Higher levels provide more abstract descriptions than 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 that are aggregations of data types. The procedural abstraction mechanisms of SLX, however, go well beyond C, and are *extremely* powerful. SLX provides not only a powerful macro capability, but a statement definition capability that allows introduction of new *statements* into SLX. You can think of user-defined statements as "super-macros" that operate at the same level as SLX's built-in statements.

1.3.4 Interfacing with the Outside World

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.

1.3.5 Why not just build a class library for simulation in C++?

If you don't already know C++, you'll find that it's a large, complex language that 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 over 100,000 lines per second on modern PCs. If you correct an error in a 50,000-line SLX model, you can be conducting your next test in half a second. How long do you think it takes a typical C++ system to compile and link a 50,000-line program?

1.4 Influences on SLX

The text above described in part, the influence of C and C++ on the design of SLX. Table 1-1 elaborates on this discussion and describes some of the other influences on SLX, in order of their importance.

Language	Influences on SLX
GPSS/H	The GPSS language first became available in 1962. It has been used worldwide for over 40 years. The list of languages for which this claim can be made is short indeed. The developers of SLX have over thirty years of experience developing and supporting GPSS/H, Wolverine Software's version of GPSS. The most important contribution of GPSS to the simulation community was the introduction of the so-called *transaction-flow* world view. That world view has been embraced by a wide variety of simulation tools and is the dominant world view in discrete event simulation. Much of the early development of SLX consisted of analysis of the strengths and weaknesses of GPSS/H. Among the strengths of GPSS/H are any easy-to-use world view and exhaustive run-time error checking. (For example, GPSS/H performs bounds checks on all array references.) Among the weaknesses of GPSS/H were (1) relatively poor data manipulation capabilities, (2) limited procedural capabilities, e.g., no subroutines, and (3) poor "factoring" of features (described below). SLX addresses the first of these deficiencies by incorporating many of C's capabilities for manipulating data. For example, GPSS/H lacks pointers. SLX includes pointers, but in GPSS/H-like fashion, all pointer assignments and references are validated at run-time. For example, NULL pointer references, the bane of all C programmers, are trapped. SLX addresses the second deficiency by incorporating most of C's features for defining functions (subroutines). The marriage of C-like functions and the transaction flow world view was far more difficult than you might think. For example, in GPSS/H an ongoing activity can execute a SPLIT, resulting in two independently scheduled activities. Consider an activity that calls a function and within that function performs a SPLIT. Which activity should be allowed to return from the procedure? The answer is that both can. SLX addresses the third deficiency by providing kernel-level functionality for essential functions only and building higher-level functionality on top of lower-level primitives. The structure of SLX is very hierarchical, while GPSS/H is very monolithic. Furthermore, in SLX, many concepts can be "factored." For example, GPSS/H has servers, and it has arrays, but no arrays of

	servers, and the collection of servers used in a program is fixed once the program begins executing. In SLX, it is possible to have arrays of servers, sets of servers, and dynamically created and destroyed servers. While SLX's servers and GPSS/H's servers, per se, are quite similar, there's no comparison in the two language's ability to organize collections of servers. Finally, SLX's kernel-level primitives support not only GPSS/H's transaction-flow world view, but a wide variety of other world views.
C	C provided a shining example of a language that had extremely high "bang for the buck." C is a very small, but extremely powerful language. Wherever appropriate, SLX borrowed C syntax and features.
Simscript	A number of SLX features were inspired by Simscript. For example, SLX's sets and picture output trace their lineage back to Simscript.
MAD	MAD (the Michigan Algorithm Decoder) was a remarkable language developed at the University of Michigan in the late 1950's. MAD was one of the earliest *extensible* languages. Its operator definition capability allowed users to develop application specific operators. For example, one could construct a collection of operators for manipulating vectors and matrices. Although SLX goes well beyond MAD's extensibility mechanisms, MAD can be considered a spiritual, if not literal, ancestor of SLX.
C++	A few C++ influences are reflected in SLX. In many cases, SLX contains features that are "watered down" in comparison to C++, but powerful enough for simulation. For example, SLX does not have constructors or destructors for its classes, but provides standard methods named "initial" and "final" that are invoked when an object is created or destroyed. The syntax of C++ is extremely compact, and in some cases, downright arcane. SLX is wordier, but more readable, and the learning curve for SLX is substantially flatter than that for C++.
Snobol	Snobol is a text-processing and pattern-matching language developed in the 1960's. A handful of SLX's macro processing pattern-matching capabilities are reminiscent of Snobol.

Table 1-1 – Influences on SLX

1.5 Conventions Used in this Documentation

SLX source code is shown in an Arial typeface.

Keywords that must be supplied as is are shown as **Boldface Arial** text.

The "|" separator is used to indicate that exactly one option must be chosen from a list of two or more options so separated. Square brackets [ ] are used to enclose optional components. Curly braces { } are used to indicate grouping in certain cases where | and [ ] might otherwise cause confusion. For example,

shout { **hey** [**you**] } | **stop** *something* ;

specifies that **shout** must be followed by **hey**, **hey you**, or **stop** followed something that you "plug in." So, the allowable forms described by the syntax above are:

shout hey ;

shout hey you ;

shout stop x ;

Curly braces followed by an asterisk ("*") enclose a specification or group of specifications that can occur as many times as required, including zero times. This notation is frequently used for lists of options that can occur in any order. For example,

shout { **hey** [**you**] | **stop** }* ;

specifies that **shout** can be followed by nothing; or **hey**, **hey you**, or **stop**, each repeated as many times as required. The following examples conform to the syntax above:

shout hey ;

shout stop hey stop hey you;

shout ;

The notation "…" is used for an item that can be repeated.

The notation ",…" is used for an item that can be repeated, with multiple items separated by commas. If only one item is given, no comma should be supplied.

1.6 The Remaining Chapters of this Book

- Chapter 2 describes general (non-simulation) features the SLX language. If you know C or C++, you'll be able to read this chapter quickly.
- Chapter 3 provides an introduction to SLX's higher-level modeling capabilities. An overview of process modeling, random variate generation, and automatically collected and reported statistics is presented in a sequence of queuing models of increasing complexity.

- Chapter 4 provides an introduction to SLX's lower-level modeling capabilities. A sequence of models of increasing complexity is presented. Some of the techniques presented in Chapter 3 are revisited, supplying implementation details deliberately glossed over in Chapter3.
- Chapter 5 describes SLX's debugger and model development environment.
- Chapter 6 describes SLX's random variates generation and statistics collection facilities.
- Chapter 7 describes SLX's extensibility mechanisms.
- Chapter 8 presents advanced high-level modeling techniques.
- Chapter 9 presents advanced low-level modeling techniques.
- Chapter 10 describes special topics such as the command line version of SLX and linking to C/C++ routines, as well as SLX in conjunction with the internet.

1.7 References

Henriksen, J. O. SLX, The Successor to GPSS/H. In *Proceedings of the 1993 Winter Simulation Conference,* ed. G.W. Evans, M. Mollaghasemi, E. C. Russel and W.E.Biles, pp. 263-268

Henriksen, J. O. , An Introduction to SLX. In *Proceedings of the 1995 Winter Simulation Conference*, ed. C. Alexopolous, pp. 502-507

Henriksen, J. O. , An Introduction to SLX™. In *Proceedings of the 1996 Winter Simulation Conference*, eds. J.M. Charnes, D.M. Morrice, D.T. Brunner, J.J. Swain , pp. 468-475

Henriksen, J.O., An Introduction to SLX. In *Proceedings of the 1997 Winter Simulation Conference*, eds. S Andradóttir, K.J. Healy, D.H. Withers, and B.L. Nelson. 559-566

Henriksen, J.O. and Th. Schulze, Einführung in SLX – den GPSS/H Nachfolger. In *Simulation and Animation '97*. Tagungsband der Otto-von-Guericke Universität Magdeburg, pp. 397 - 418

Henriksen, J.O, F. Preuß and Th. Schulze, Simulation des 5-Philosophen-Problem in SLX . In *Tagungsband zum 10. Symposium Simulationstechnik 1996*, Fortschritte in der Simulationstechnik, ed. Wilfried Krug, Vieweg 1996

Henriksen, J.O. SLX™ and Animation™: Improved Integration Between Simulation and Animation . In Proceedings der Tagung „Simulation and Animation '97", ed. O.Deussen and P. Lorenz, SCS Int. 1997, pp.287-294

Henriksen, J. O. , An Introduction to SLX™. In *Proceedings of the 1997 Winter Simulation Conference*, eds. S. Andradoittir, K. J. Heally, D. H. Withers, and B. L. Nelson, pp. 559-566

Schriber, T. J. and D. T. Brunner, Inside Software: How Ist Works and Why It Matters. In *Proceedings of the 1997 Winter Simulation Conference*, eds. S. Andradoittir, K. J. Heally, D. H. Withers, and B. L. Nelson, pp. 14-22