Proceedings of the 1996 Winter Simulation Conference
ed. J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain

AN INTRODUCTION TO SLX™

James O. Henriksen

Wolverine Software Corporation
7617 Little River Turnpike, Suite 900
Annandale, VA 22003-2603, U.S.A.

ABSTRACT

SLX is Wolverine Software's "next generation" simulation language. SLX builds on the strengths of Wolverine's GPSS/H (Henriksen & Crain, 1996). It provides powerful simulation capabilities in a modern, C-like language framework. SLX provides a multiplicity of layers, ranging from the SLX kernel, at the bottom, through traditional simulation languages, e.g., GPSS/H, in the middle, to application-specific language dialects and extensions at the top. This paper provides an overview of SLX for readers who are already familiar with simulation. An earlier paper (Henriksen 1995) presented key concepts of the architecture of SLX. As of this writing, SLX has been heavily used by a number of alpha testers. Insights gained from our interactions with the software testers will be presented.

1 INTRODUCTION

The most important characteristic of SLX is its layered architecture. The success of SLX's layered approach depends on several factors:

A. The layers are well-conceived. In developing SLX, we have had the luxury of drawing on years of experience with GPSS/H. A great deal of SLX is based on GPSS/H. In some cases, source code from GPSS/H has been directly "lifted" for use in SLX. In other cases, we have modified, simplified, or adapted GPSS/H algorithms. In a few cases, we have eliminated pitfalls and shortcomings of GPSS/H. The end result is an extremely well-designed system.

B. The layers are not too far apart. Many other languages provide multiple layers, but typically there are wide gulfs between the layers. For example, a language 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. One must become familiar with many details of the C or FORTRAN *implementation* of the simulation language to be able to add C or FORTRAN extensions. Even worse, virtually none of the error checking and other safeguards provided in the simulation language are available in C or FORTRAN. The SLX kernel language is a powerful, C-like language, so users of SLX almost never find it necessary to drop down into a lower-level, more powerful language. Furthermore, the SLX kernel language includes complete checking to prevent errors such as referencing beyond the end of an array and using invalid pointer variables. The layers above the SLX kernel exploit kernel capabilities in straightforward ways. Transitions from layer to layer are very smooth.

C. The mechanisms for moving from layer to layer are very powerful. These mechanisms are *abstraction* mechanisms. A "higher level" entity provides a more abstract description than a "lower level" entity. 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 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.

In the sections which follow, selected features of the SLX kernel are presented. Following the presentation of the SLX kernel, SLX's extensibility mechanisms are illustrated. Finally, brief presentations of how SLX has been used and how SLX is taught are presented.

2 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 Section 2, we will present some representative features, to illustrate the functionality, ease-of-of-use, and *ease-of-learning* of SLX.

2.1 Objects and Pointers to Objects

The very mention of the word "object" inspires a wide range of expectations and emotions, due to the widespread influence of object-oriented programming (OOP). SLX has been influenced by OOP and incorporates some OOP ideas, but *SLX is not a truly object-oriented language*. Such a statement is a rarity in this day. Many products claiming to be object-oriented are far from it. In fact, although we do not claim that SLX is object-oriented, it is probably more so than some products for which OOP architecture is claimed.

In SLX, objects are used in two ways. *Passive* objects are used for modeling entities which have no "executable" behavior. For example, a parking lot could be modeled as a passive object. GPSS/H Facilities, Queues, and Storages are implemented as passive SLX objects. *Active* objects have executable behavior patterns. Customers in a supermarket are a good example of entities that would probably be modeled as active objects. SLX active objects are roughly equivalent to GPSS/H transactions. 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, the 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.

SLX objects can have a number of *standard properties*. All standard properties are comprised of explicitly identified sections of executable code. The *initial* property is invoked when an object is created. The *final* property is invoked when an object is about to be destroyed. The *report* property is invoked by the *report* statement; it is used for the obvious purpose. The *actions* property specifies the behavior pattern for an active object. It is invoked by the *activate* statement (discussed in the next section). A sample object follows:

```
object customer
  {
  integer  number_of_items;

  initial
    {
    number_of_items =
    rv_uniform(stream1, 10, 20);
    }

  actions
    {
    ...
    }

  report
    {
    print (ME, number_of_items)
    "* exiting, items purchased: *\n";
    }
  };
```

Objects are created by using the *new* operator, which returns a pointer to the newly created object:

```
pointer(customer)       cust;
cust = new customer;
```

The initial property of the customer is executed before the pointer to the new customer is assigned to `cust`. The initial property can be thought of as an easier-to-use counterpart to a C++ constructor. All uses of pointers are validated to ensure that pointers always point to objects of the proper type (or contain a NULL value). Use counts are maintained for all objects. If pointer1 and pointer2 are pointer variables, an assignment statement of the form "pointer1 = pointer2" causes the use count for the object pointed to by pointer1 to be decremented and the use count for the object pointed to by pointer2 to be incremented. Storage for an

object is released if and only if its use count goes to zero.

2.2 Active Objects

An active object has an *actions* property. When the *activate* verb is used, a *puck* is created for the object and placed on the active puck list; i.e., the puck is placed in a ready-to-execute state. Pucks are the schedulable entities in SLX. Scheduled time delays and state-based delays, e.g., waiting for a server to become available, are puck-based operations. Thus manipulation of pucks is the basic mechanism by which a collection of objects experiences events over time. Pucks embody the means of achieving simulated parallelism.

Two kinds of parallelism can be described in SLX. Large-scale parallelism consists of interactions among active objects. For example, in a model of a freeway toll booth system, active car, truck, and bus objects would all interact. Small-scale or *local* parallelism consists of parallel activities carried out by the same object. For example, a customer entering a bank may decide that (s)he will wait in line for no longer than two minutes. At the end of that time, if the customer has not been served, (s)he will renege (exit the system without having been served.)

At first glance, the distinction between large-scale and local parallelism might appear to be somewhat arbitrary. What's large-scale, and what's local? In SLX, a precise distinction is made based on the SLX verbs which are used. Objects are activated by using the *activate* verb. In most cases activate is used in conjunction with new:

```
activate new customer;
```

The above statement creates a new customer object, creates a puck for it and places the puck in the active puck list, poised to execute the first statement in the customer object's actions property. The puck which executes the "activate new" statement continues executing. The new customer will compete with other pucks (according to their respective priorities) after the current puck has gone as far as it can in the actions property at the current instant of simulated time.

Local parallelism is described by using the *fork* statement:

```
fork
  {
  offspring actions
  }
parent
  {
  parent actions
  }
```

Execution of the fork statement creates an additional puck for the currently active object. The newly created puck is placed in the active puck list, poised to execute the offspring actions clause of the fork statement. The parent puck executes the optional parent actions clause. Unless otherwise specified, both pucks continue execution with the next statement following the fork statement. An example of the fork statement is given in Section 2.4.

2.3 Time Advance

Time advance is provided by the advance statement, modeled on the ADVANCE block of GPSS/H, e.g.,

```
advance rv_expo(stream2, 10.0);
```

The advance statement removes the puck from the active puck list and places it on the future event list, scheduled to resume execution after the specified time delay has taken place.

2.4 Control Variables & Wait Until

In SLX, state-conditioned 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.

Let us consider an example which illustrates the use of the fork statement in conjunction with *wait until*. Assume that customer objects flowing through a model reach a point where they are willing to wait a maximum of two minutes for service. If they are not served within two minutes, they exit the system; i.e., they renege.

```
object customer
  {
  control boolean reneged;
  actions
    {
    ...
    fork
      {
      advance 2.0;    // max wait time
      reneged = TRUE;
      terminate;
      }
    parent
      {
      wait until (reneged
         or server available);
      if (reneged)
      terminate;
      }
    ...
    }
  };
```

In the above example, a Boolean control variable is used within the customer object for communicating "renege" status between two pucks which share the same object. At the point at which the customer begins waiting for service, it forks, creating a second puck. The offspring puck executes the logic enclosed in braces immediately following the fork statement. The original puck executes only the logic contained within braces following the "parent" clause. The offspring puck undergoes a two-minute delay, sets reneged to TRUE, and terminates itself. The parent puck waits for either the server to become available or for the two minutes to elapse. When it comes out of the `wait until`, it must distinguish which of these two possibilities has taken place. If the two minutes have elapsed, reneged will be TRUE, and the parent puck will terminate itself. If not, the parent will continue executing.

The sharing of a single data area makes communication between the two pucks trivial. The "reneged" variable shared by the two pucks is all that is needed to accomplish the communication required in this example. Note that if there are multiple customers active at a given time, each customer will have its own data area, so the "reneged" status for one customer cannot be confused with that of another.

In many simulation languages, operations such as reneging are difficult to implement. Because of this, languages sometimes include operations such as reneging as built-in features. Unfortunately if the language designer's concept of reneging does not exactly match your requirements, you're stuck. In SLX, carefully designed rock-bottom primitives allow you to build your own capabilities if none of the ones provided by others meet your needs.

2.5 Sets

SLX includes the capability for defining and manipulating sets of objects. Sets can be homogeneous (comprised of objects of a single type) or universal (comprised of objects of arbitrary types.) Homogeneous sets can be ranked in ascending or descending order on one or more attributes of objects of the type comprising the set. Some examples of set definitions follow:

```
set(widget) ranked FIFO  fifo_set;
set(widget) ranked LIFO  lifo_set;;
set(*)  universal_set;
set(job) ranked(ascending due_date,
descending priority)   job_set;
```

An iteration construct is provided for looping through the members of a set:

```
pointer(widget) w;
for (w = each widget in fifo_set)
  {
  ...
  }
```

If the subject set is a universal set, `for each` selects only objects of the specified type, skipping over any objects which are of other types. Arbitrarily complex forms of iteration can be built from lower-level first, last, successor, and predecessor primitives:

```
w = first widget in widget_set;
w = last widget in widget_set;
w = successor(w) in widget_set;
w = predecessor(w) in widget_set;
```

Using these primitives, *any* form of set iteration can be achieved.

3 EXTENSIBILITY FEATURES

SLX was designed to be an extensible platform on which a wide variety of higher level simulation applications could be built. In this section we will briefly present an example of how the extensibility mechanisms can be used to build new features out of old ones.

Suppose we wish to build a simple telephone book. Furthermore, assume that each entry in the book contains only a first name, a last name, and a telephone number. An entry in the phone book can be described as an SLX object:

```
object  book_entry
  {
  fstring(20)     first_name,
                  last_name;

  int             phone_number;
  }
```

Suppose that we wish to retrieve entries from the phone book both by name and by number. We could construct two SLX sets, one ranked by name and one ranked by number:

```
set(book_entry)ranked(ascending
  last_name, ascending first_name)
    phone_book;

set(book_entry) ranked (ascending
  phone_number)
    reverse_phone_book;
```

The name-sorted phone book could be printed as follows:

```
pointer(book_entry) name;
for (name = each book_entry in
  phone_book)
    print( name -> last_name,
           name -> first_name,
           name -> phone_number)

"*****, *****: *******\n":
```

Of course, the printing of phone books is best left to the telephone company. For the rest of us, the most common use of the phone book is to look up numbers. For the police, a relatively common use is to look up the name associated with a number. The SLX kernel contains no "look up" primitive; however, one can be constructed easily from existing language features. The following approach can be used to look up a phone number, given first and last names. First, copy the first and last names into a dummy book_entry object. We call this object query_book_entry; Next, place the query_book_entry object into phone_book. Since phone_book is ranked by ascending last and first name, and entries with identical keys are inserted into a set in FIFO order, we know that query_book_entry will be placed into phone_book immediately following the entry we wish to look up. Thus, the desired entry will be the predecessor of query_book_entry in phone_book. If the name we are looking up is not in the phone book, either the predecessor of query_book_entry will be NULL, or it will contain the wrong names. These conditions are easily detected. Finally, the query_book_entry can be removed from phone_book.

While each of the steps outlined above is straightforward, we'd like to have a less cumbersome way of issuing queries. SLX provides a statement definition facility which allows us to construct a "retrieve" statement. The definition of our retrieve statement is shown in Figure 1. The first line of the definition is a prototype which specifies the components of the retrieve statement. Names preceded by a pound sign ("#") represent components that are supplied by the user for each use of the statement. Braces ("{}") are used to enclose a group of specifications. The notation ",..." indicates that the immediately preceding component can be repeated as many times as desired, with repetitions separated by commas. The "@" in front of the "from" keyword tells SLX to ignore the usual meaning of "from" and treat it as a keyword of the retrieve statement. ("from" is a reserved word in SLX.)

The definition section specifies the mapping of the retrieve statement into lower-level SLX statements. Within the definition section, the *expand* statement is used to specify the lower-level SLX code that is to carry out the retrieval operation. The expand statement specifies one or more lines of output which is injected into the SLX compiler's input stream. A list of expressions can be supplied to be edited into the generated lines of output. Within an output line, groups of adjacent "#" symbols are replaced by edited values.

With one very important exception, this approach is similar to the use of *macros* in many languages. In most languages, the statements which are available to specify the internal logic of a macro are either very limited and use a syntax different from the host language, or they comprise a comparatively weak subset of the host language. In SLX, the "macro language" is SLX itself. Only a handful of statements are excluded from use within an SLX statement definition. For example, simulation constructs such as *wait until* or *advance* have no meaning during compilation of a program. Apart from these obvious restrictions, most of the rest of the SLX language can be used. For example, it is even possible to read a file as part of the process of expanding a statement!

The example in Figure 1 makes use of a local integer variable, i, which is used to iterate through the list of comma-delimited "#key = #value" items. The iteration is terminated when an empty value of #key[i] is encountered. This is an SLX convention. If a list of N items is provided, #key[1...N] will have non-empty values, and #key[i] will be empty for i > N.

A sample invocation of the retrieve statement and its expansion is shown in Figure 2. This example illustrates the power of SLX's statement definition facility. The retrieve statements are easy to read. If you didn't know that the retrieve statement was built using the statement definition facility, you would probably think that it was a built-in SLX statement. This example illustrates the extensibility of SLX. It's very easy to reshape the language by adding new statements which are specific to the kinds of problems you are working with. Whether you need a capability which is not present in basic SLX, or whether you just want a more expressive way of specifying complex logic in a more compact form, the statement definition facility can be of great use.

```
statement retrieve #ptr = #otype ( { #key = #value },... ) @from #set ;
    definition
        {
        int     i;

        expand                              "{\n";

        for (i=1; #key[i] != ""; i++)
                expand(#otype, #key[i], #value[i])

                                            "query_#.# = #;\n";

        expand(#otype, #set)                "place &query_# into #;\n";

        expand (#ptr, #otype, #set)         "# = predecessor(&query_#) in #;\n";

        expand(#otype, #set)                "remove &query_# from #;\n";

        expand(#ptr, #ptr, #key[1], #value[1])
                                            "if (# != NULL)\n  if (# -> # != #";

        for (i=2; #key[i] != ""; i++)
                expand(#ptr, #key[i], #value[i])

                                            " or # -> # != (#)";

        expand(#ptr)                        ")\n     # = NULL;\n";
        expand                              "}\n";
        }
```

Figure 1: The Definition of a "Retrieve" Statement

```
retrieve e = book_entry(last_name=lname, first_name=fname) from phone_book

    {
    query_book_entry.last_name = lname;
    query_book_entry.first_name = fname;
    place &query_book_entry into phone_book;
    e = predecessor(&query_book_entry) in phone_book;
    remove &query_book_entry from phone_book;
    if (e != NULL)
      if (e -> last_name != lname or e -> first_name != (fname))
         e = NULL;
    }
```

Figure 2: A Sample Expansion of the "Retrieve" Statement

4 EXPERIENCE WITH SLX TESTERS

As of this writing SLX has been intensively used by a select group of testers around the world. Interacting with the testers has resulted in a number of improvements to SLX and has yielded some interesting insights into how SLX is (and will be) used. The projects tackled by SLX testers have all been fairly intense efforts. Included among these have been a half-dozen masters degree thesis projects and one very large scale, real-world transportation model.

Two modest surprises have come out of the testing. First, the extent to which users have made direct use of SLX kernel-level features has exceeded our expectations. Perhaps biased by our years of experience with GPSS/H, we had expected that most users would prefer to use higher-level features, only occasionally resorting to kernel-level features. Virtually all the testers have made heavy use of low-level scheduling primitives such as wait until and fork. A second modest surprise has been the extent to which the testers have exploited SLX's statement-definition capabilities. We had envisioned the statement definition capability as being of interest primarily to builders of higher-level simulation packages, but nearly everyone who has touched SLX has made use of its statement definition facility.

Taken together, these two surprises demonstrate that we have achieved our goal of extensibility. SLX is easily adapted to different purposes by different users. With the ability to incorporate new statements into the language, the extended language used by the developer of a package for manufacturing applications is quite different from the extended language used by someone tackling transportation problems. Yet the two are built on a common foundation, and the underlying software is hardened and tempered by exposure to widely differing usage patterns.

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. 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 (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.

As of this writing, an introductory SLX textbook is under development. A preliminary (at least) edition of the book will be available by the time of the 1996 Winter Simulation Conference. The book will use the approach outlined above for simultaneously exposing fundamental simulation concepts and their implementation in SLX. Given the enormous popularity of graphically-based, fill-in-the-blanks modeling tools, our approach to teaching SLX entails a degree of risk. We're going back to teaching the basics at a time when much of the rest of world is moving in the opposite direction. Our approach is motivated by a year's experience with our SLX testers and over twenty years' experience in helping users learn to use our simulation software. Our experience is biased by years of exposure to people who still believe it's important to know what they're doing.

6 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. The efficacy of the approaches offered by SLX has been demonstrated through intensive use. The benefits for teaching simulation derived from being able to simultaneously expose language primitives and fundamental modeling concepts remain to be demonstrated as of this writing. However, we are excited by SLX's great potential.

REFERENCES

Henriksen, J.O. 1995. An Introduction to SLX. In *Proceedings of the 1995 Winter Simulation Conference*, ed. C. Alexopoulos. 502-509. Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

Henriksen, J.O., and R.C. Crain. 1996. *GPSS/H reference manual*, fourth edition. Annandale, VA: Wolverine Software Corporation.

AUTHOR BIOGRAPHY

JAMES O. HENRIKSEN is the president of Wolverine Software Corporation. He is a frequent contributor to the literature on simulation and has presented many papers at the Winter Simulation Conference. Mr. Henriksen served as the Business Chairman of the 1981 Winter Simulation Conference and as the General Chairman of the 1986 Winter Simulation Conference. He has also served on the Board of Directors of the conference as the ACM/SIGSIM representative.