*Proceedings of the 1995 Winter Simulation Conference*
*ed. C. Alexopoulos, K. Kang, W. R. Lilegdon, and D. Goldsman*

# AN INTRODUCTION TO SLX

James O. Henriksen

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

## ABSTRACT

This paper provides introduction to SLX (Henriksen 1993) for readers who are already familiar with simulation. Comparisons with GPSS/H (Banks, Carson and Sy 1989; Henriksen and Crain 1989; Schriber 1991; and Smith, Brunner and Crain 1992) are used to provide a frame of reference for describing SLX features. The goal of the SLX project is to produce a simulation system which provides a multiplicity of layers, ranging from the SLX kernel, at the bottom, all the way up to layers which provide graphical model building "without programming." In this paper, only the "lower" layers of SLX are described. Accordingly, this paper will perhaps be of greater interest to simulation software package developers than to end users of simulation software. Six key concepts which underlie the SLX kernel are presented, and SLX's extensibility mechanisms, which facilitate the construction of higher layers from lower layers, are illustrated.

## 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 virtually 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, the six key concepts underlying the architecture of the SLX kernel are presented. Following the presentation of the SLX kernel, SLX's extensibility mechanisms are illustrated. Finally, a version of GPSS/H hosted in SLX is described.

## 2 THE SIX KEY CONCEPTS IN THE ARCHITECTURE OF THE SLX KERNEL

### 2.1 Key Concept #1 - The Large Building Blocks (Modules, Procedures, and Objects)

#### 2.1.1 Modules

The largest building block in SLX is the *module*. Every SLX program is comprised of at least one module, and all but the very simplest will have at least two - one containing your code, and one (or more) contained in an *include* file provided by someone else. The module is the largest unit of *encapsulation* in SLX. Encapsulation allows selective specification of the "visibility" outside a module of entities defined within the module. Entities defined within a module are visible throughout the module. Their visibility outside the module is under the control of the module's developer. For example, variables can be public or private (visible or invisible, respectively, outside the module), read-only (readable but not modifiable outside the module), or write-only. In the SLX-hosted implementation of GPSS/H, many entity attributes implemented as SNAs (Standard Numerical Attributes) in GPSS/H are implemented as public, read-only variables in SLX. For example, the current length of a Queue is so defined. This allows a model to directly access the length of the Queue; however, the model can modify the length of the queue only by using code provided for that purpose.

Developers of modules can use encapsulation for two purposes: (1) to protect users of the module from modifying data they should not be allowed to change, and (2) to hide proprietary implementation information.

#### 2.1.2 Procedures

The second largest building block in SLX is the procedure. For the most part, procedures in SLX are like procedures (which may be called functions or subroutines) in other languages. The SLX compiler "sees" all procedure definitions and rigidly enforces agreement between the arguments of procedure invocations and the formal parameters of procedure definitions. Although the syntax of SLX is largely modeled on that of C, procedure definitions are considerably different in SLX. The SLX syntax is wordier, and the SLX semantics are more rigid. (You won't find procedures with a variable number of arguments in SLX, for example.)

#### 2.1.3 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 machinist in a job shop. In selecting the next job to be performed, (s)he may consider a number of parameters, such as job priority, due date, resource requirements, etc. Often, it is easiest to model such behavior using active objects.

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). The *clear* and *reset* properties specify what should be done to an object when statistics are cleared or reset. (These properties provide lower-level support for higher-level verbs such as GPSS/H's CLEAR and RESET statements, which are used to define statistics collection in a sequence of experiments.)

### 2.2 Key Concept #2 - Activation Records

Data for SLX modules, procedures, and objects is stored in activation records (A/Rs). The A/Rs of modules are allocated when an SLX program is loaded. At the start of execution, module A/Rs are initialized, and an A/R for the main procedure is allocated and initialized. All of this takes place before control passes to the first "executable" statement of the main procedure. For procedures other than the main procedure, i.e., subroutines and functions, an A/R is created when the procedure is called and (almost always) destroyed when the procedure returns to its caller.

Arguments to a procedure are placed into the A/R prior to the call. If a called procedure returns a value, i.e., if the procedure is a "function," the calling procedure places the target address of the result in the called procedure's A/R.

The A/Rs for objects are created in two ways. Objects can be created explicitly, by means of the *new* operator, or implicitly, by being declared as "local" objects in a module or procedure. The locations of objects created explicitly are usually kept track of by using *pointer* variables:

```
pointer(widget)    joe, fred;
joe = new widget;
fred = joe;
```

Implicitly created objects are stored in the A/R of the procedure or module which contains them. Their declarations appear similar to those of other variables:

```
integer    i;
float      x;
widget     joe;
```

Although not apparent, the creation of an A/R is extremely complex beneath the surface. Storage must be allocated for the A/R (dynamically, in the case of explicit creation, and statically, as part of another A/R, in the case of implicit creation.) All variables in an A/R are set to known initial values. For example, integers and floats are set to zero, and pointers are set to the value NULL, which means they point to "nothing."

Objects which are implicitly created are destroyed when the A/R in which they reside is destroyed. Objects which are explicitly created can be explicitly destroyed by using the *destroy* statement:

```
pointer(widget)    joe;
joe = new widget;
...
destroy joe;
```

Note that destroy operates on a *pointer*. It releases the storage of the object pointed to by the pointer and assigns a NULL value to the pointer.

Before the A/R of an object or procedure can be destroyed, a check must be made to verify that the A/R is no longer "in use," i.e., that no pointers still point to any part of it. For this purpose, *use counts* are maintained for all A/Rs. When an attempt is made to destroy an A/R with a nonzero use count, the destruction is deferred until such time (if ever) the use count goes to zero. This prevents potentially disastrous "gone but not forgotten" bugs which can happen in languages like C. In C, pointers to released storage can be used to fetch unknown values, or far worse, to store into unknown areas of memory. Use counts also prevent "forgotten but not gone" problems. When the use count of an A/R goes to zero, the A/R is automatically released. Consider the following example:

```
pointer(widget)    joe, fred;
joe = new widget;
fred = new widget;
joe = fred;
```

When `joe` is assigned the value of `fred`, the use count of the widget it pointed to prior to the assignment of a new value goes to zero, so the A/R for the object is released.

### 2.3 Key Concept #3 - Active Objects & Pucks

An active object is distinguished from a passive object by its *actions* property:

```
object fork_lift
    {
    variable definitions...
    actions
      {
      (statements specifying the forklift's behavior)
      }
    };
```

The actions are invoked by means of the *activate* operator, which operates on pointers to objects. Activate is usually applied immediately to a *new* operator:

```
activate new fork_lift;
```

Activation and creation can, however, be performed as distinct steps:

```
pointer(fork_lift) f;
f = new fork_lift;
...
activate f;
```

When the latter form is used, it is usually for purposes of performing actions on the object before it is activated.

Active objects are destroyed when they execute a *terminate* statement.

Activating an object creates what is called a *puck* for the object. What is a puck? Pucks are the "schedulable" entities in an SLX model. When an object is activated, its puck is placed on the current events chain (CEC). The CEC is a list of all pucks eligible to execute at the current instant of simulated time. Like GPSS/H's CEC, the SLX CEC is ordered by puck priority, and for pucks of equal priority, on a FIFO basis.

When a scheduled time delay, e.g., a service time or interarrival time, occurs in a model, it is the puck which is actually scheduled. If an object must wait for a given state to arise, e.g., for a requested server to become available, it is the puck which waits. When a puck must wait, it is removed from the CEC and placed on another chain.

For a given object, additional pucks can be created by executing *fork* statements. Pucks created by fork statements share the same activation record. The fork statement is a very convenient way for modeling local, "small scale" parallelism for an object. An example of this use is given in Key Concept #5, below. For less local, "large scale" parallelism, it is usually best to create active objects of another type, to model system behavior in terms of interactions *between* objects.

This section has introduced the concept of SLX pucks. In each of the three following sections, the critical role of the puck is further explained.

### 2.4 Key Concept #4 - Scheduled Time Delays

In a simulation model, two kinds of delay can take place, delays of scheduled duration, and delays which are conditioned on some component of the model reaching a specified state, e.g., a server becoming available. Scheduled delays are described in this section, and state-conditioned delays are described in the next.

Scheduled time delays are modeled by use of the *advance* statement, e.g.,

```
advance  rv_expo(stream1, 10.0);
```

When a puck executes an advance, the puck is removed from the CEC and placed on the Future Events Chain (FEC.) The roles of the CEC and FEC in SLX are similar to the CEC and FEC of GPSS/H. When all pucks have been removed from the CEC at any given instant of simulated time, the simulator clock is updated to the next imminent event time, the lowest scheduled "move time" for the puck(s) on the FEC. As part of the clock update, all pucks with this move time are moved from the FEC to the CEC. (Additional details are given in Key Concept #6.)

### 2.5 Key Concept #5 - 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 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);
```

Finally, SLX also supports *indefinite* (user-managed) waits. There are three steps 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. In SLX, GPSS/H User Chains are implemented as indefinite waits. The *link* statement places pucks into a set (the User Chain), and the *unlink* statement reactivates pucks.

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 waiting time
            reneged = TRUE;
            terminate;
            }
        parent
            {
            wait until(server_available || reneged);
            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 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 common A/R makes communication between the two pucks trivial. The "reneged" variable in the A/R 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 A/R, 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.

### 2.6 Key Concept #6 - The Architecture of the SLX Simulator

A flowchart describing the SLX simulator is shown in Figure 1. As you might expect, the flowchart describes the operation of the SLX simulator as a process of puck management. At any point in time, a puck will be in exactly one of the following states:

(1) On the Current Events Chain (CEC), eligible to execute at the current instant of simulated time. The CEC is sorted by descending puck priority, and for pucks of equal priority, on a FIFO basis.
(2) On the Future Events Chain (FEC), undergoing a scheduled time delay. The FEC is sorted by ascending puck move time, and on a FIFO basis for pucks with equal move times.
(3) On one or more reactivation chains for a wait until.
(4) None of the above. This is the state for pucks undergoing indefinite, user-managed waits, as described at the end of the previous section.

The heart of the SLX simulator algorithm is the selection of a puck to attempt to move through the model and the disposition of the selected puck. Puck selection is very simple: the puck selected is always the first puck on the CEC. Unlike GPSS/H, SLX has no "scan-inactive" pucks (which require skipping over) on the CEC. The selected puck resumes execution at the point at which it last left off. For a newly activated puck, this will be at the start of the *actions* property of its object. For a newly fork-created puck, this will be at the statement immediately following the fork.

Once a puck has been selected, it moves as far as it can through the model. Depending on circumstances, this can be a very short or very long distance. When the puck has moved as far as it can, it returns control to the SLX simulator in one of seven states:

(1) If a scheduled time delay has been issued (at an advance statement), the puck is removed from the CEC and placed on the FEC in a position determined by its scheduled move time.
(2) If the puck has been delayed at a wait until statement, it is removed from the CEC and placed on one or more reactivation chains associated with control variables used in the wait until condition.
(3) If the puck has been placed into an indefinite wait at a wait statement (with no "until" clause), it is removed from the CEC. The data structures used to keep track of the puck so that it can subsequently be explicitly reactivated are the user's responsibility.
(4) If the puck has terminated itself, i.e., executed a terminate statement, it is removed from the CEC.
(5) If the puck executes an *exit* statement, execution of the model will be stopped.
(6) If the puck causes an execution error, execution of the model will be stopped.
(7) If the puck executes a *yield* statement, it will remain on the CEC, and another puck will be selected. Note that this is the *only* one of the seven cases in which a puck meaningfully remains on the CEC. If the yield statement specifies the puck which is to be selected, the specified puck is the next one to be moved. This form of the yield statement is used to assure that when the active puck causes a certain condition to arise, the proper puck immediately responds to the condition, without regard to puck priority or other factors. If no puck is specified in the yield statement, the first puck on the CEC is selected. Note that this may be the puck which issued the yield. This form of the yield statement is usually used to allow pucks of higher priority to respond to conditions created by a puck of lower priority. This form of yield is directly analogous to the BUFFER Block in GPSS/H.

When the active puck returns in states 1-4, a next puck must be selected. If the CEC is empty, there are no more pucks able to move at the current instant of simulated time, and the simulator clock must be updated. If the CEC is non-empty, the next puck to be moved through the model will now be the first puck on the CEC.

## 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 discuss two of the extensibility mechanisms which help make this possible. Data extensibility mechanisms allow the introduction of new data "types" into SLX. Procedural abstraction mechanisms allow the introduction of new statements and operators (macros).

SLX provides two ways in which new data types can be constructed. The type definition statement defines a new data type in terms of an old one. For example,

```
type status enum(running, idle, down);
```

defines a data type named status as a C-style enumerated type.

The second way in which new data types can be constructed is to include objects as sub-objects of larger objects. This paradigm is known as *composition*. It is in many respects the opposite of OOP's *inheritance* paradigm. When using composition, one thinks of building large components out of smaller ones. When using inheritance, one takes a top-down approach, specifying the most general classes of objects first, and implementing more specific objects as subclasses. The following is an example of composition:

```
object XY_position
    {
    float   xpos,
            ypos;
    }

object battleship
    {
    XY_position     ship_loc;
    ...
    }

pointer(battleship) bs;

bs = new battleship;
bs -> ship_loc.xpos = ...
bs -> ship_loc.ypos = ...
```

Perhaps the most impressive extensibility mechanism of SLX is its statement definition facility, which allows the introduction of new *statements* into the language. There are three major components of a statement definition:

(1) a prototype which specifies the syntax of the statement (informally, "how it looks");
(2) optional logic and looping within the definition, responding to the presence, absence, and other characteristics of statement components; and
(3) one or more *expand* statements which inject "generated" text into the source stream seen by the SLX compiler.

Subsequent invocations of defined statements are replaced by the text generated according to their definitions.

Statement prototypes employ a small number of metasymbols to describe statement syntax. For example, braces ("{}") are used to enclosed groups of mandatory components, brackets ("[]") are used to enclose groups of optional components, and ",..." is used to specify a comma-delimited list. Metasymbols occur only in the prototype, but not in invocations of the statement. The following is a prototype for the SLX enter statement, based on the GPSS/H ENTER Block:

```
statement enter #storage [units=#inc];
```

This prototype says that #storage is a mandatory component, and #inc is an optional component, which if used, is specified as "units=*expression*".

The following are valid invocations of the enter statement:

```
enter  joe;
enter  fred, units=x+y;
```

Suppose we wish to define a new form of the GPSS/H SEIZE Block which permits specification of a list of Facilities, all of which are to be SEIZEd in sequence. The following prototype would do the job:

```
statement seize { ALL(#list,...) | #facility };
```

This prototype says that the seize statement has one mandatory component, whose specifications are enclosed between braces ("{ }") in the prototype. That component can take one of two forms. The two forms are separated by a vertical bar ("|"). The first form consists of the word ALL, followed by a parenthesized list of one or more expressions. If more than one expression is supplied, expressions must be separated by commas. Finally, the second form is specified as a simple expression.

The following are valid invocations of the seize statement:

```
seize joe -> server; // second form,
seize ALL(joe, bill, fred);   // first form
```

The complete definition of the seize statement is as follows:

```
statement  seize { ALL(#list,...) | #facility };
    definition
        {
        integer i;

        if (#facility != "")
          expand(#facility) "SEIZE(#);\n";
        else
          {
          expand "{\n";

          for (i = 1; #list[i] != ""; i += 1)

              expand(#list[i]) "SEIZE(#);\n";
          expand "}\n";
          }
        }
```

The definition of the seize statement contains a local integer variable, *i*, which is used to iterate through the list of Facilities, if the ALL clause is used.

The two forms are distinguished by asking whether #facility is an empty string. The SLX compiler performs pattern matching between the invocation of a statement and its prototype. If the ALL form is used, #facility will be empty. Conversely, if the simple form is used, #list will be empty.

If the simple form is used, the seize statement expands into a call of the procedure named SEIZE, where the expression supplied as #facility is plugged in (by the *expand* statement) as the argument to the procedure. Thus,

```
seize(joe);
```

expands into

```
SEIZE(joe);
```

If the ALL form is used, the SLX compiler plugs the individual expressions used into #list[i], for as many items as are specified. Beyond the last item, #list[i] will be empty. The definition simply loops through #list[i], generating a call of SEIZE for each Facility in the ALL list. Thus,

```
seize ALL(joe, bill, fred);
```

expands into

```
{
SEIZE(joe);
SEIZE(bill);
SEIZE(fred);
}
```

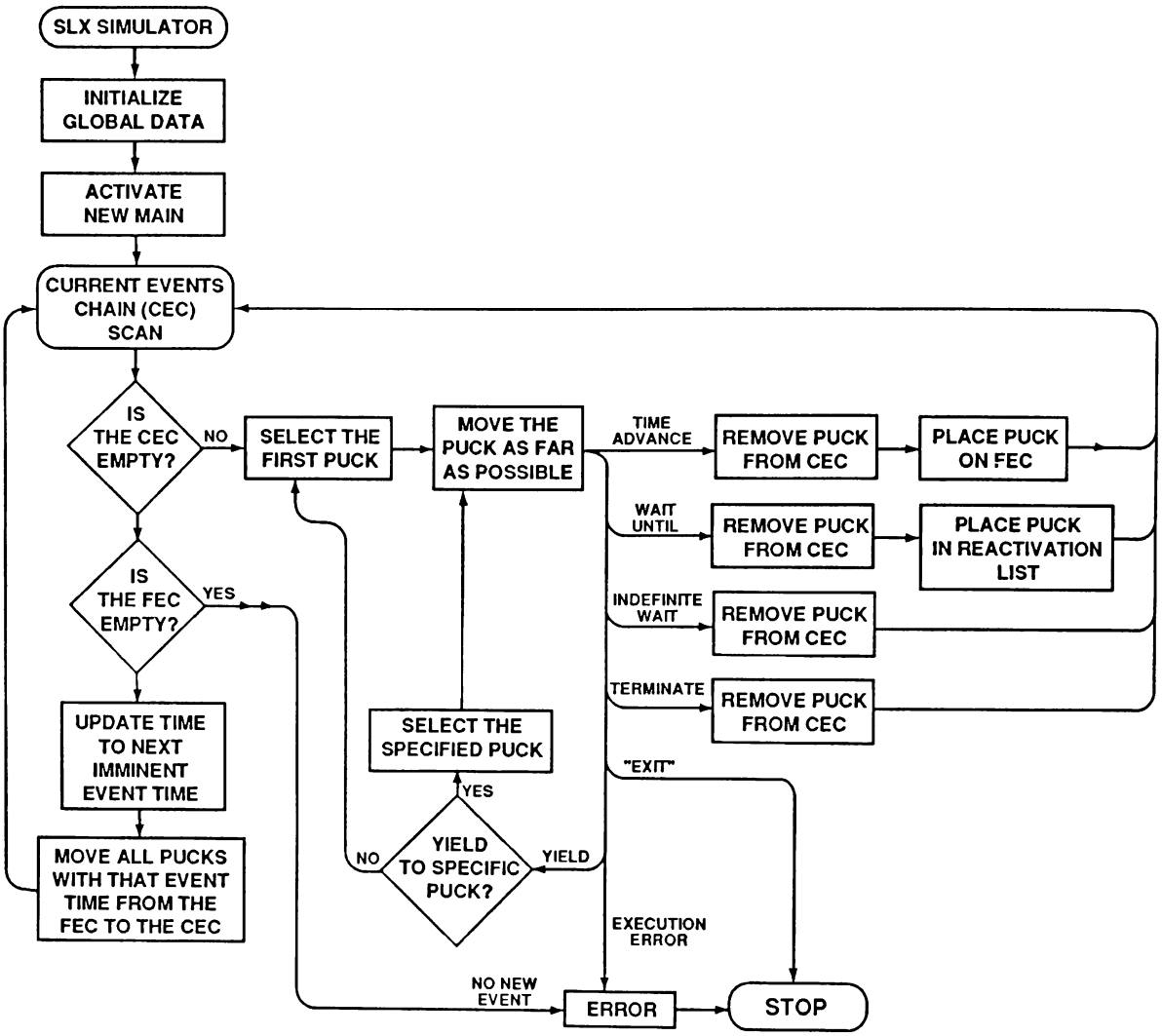


Figure 1 – The SLX Simulator Algorithm

Note that the enclosing braces are necessary to assure that the collection of generated statements is treated as a single, individual group. If they were omitted, the following would yield unexpected results:

```
if (condition)
    seize ALL(bill,ralph);
```

would result in

```
if (condition)
    SEIZE(bill);
    SEIZE(fred);
```

Only the first SEIZE would be controlled by the if statement. This would be a gross violation of the law of least astonishment (things ought to work in the way they're expected to.)

In the foregoing discussion, we made an important distinction between procedures and statements which ultimately invoke procedures. Using SLX statement definitions, rather than just using procedure calls offers two distinct advantages. First, readability is enhanced by eliminating the extra punctuation required for a procedure call. One can readily see that

```
seize joe;
```

is a far prettier notation than

```
SEIZE(joe);
```

Second, in the case of GPSS/H statements more complex than SEIZE, statement definitions can handle optional or repeated arguments and format appropriate lower-level SLX executable statements. Some languages, e.g, C, allow procedures with a variable number of arguments. Using such procedures would be an alternative to SEIZE ALL, described above. However, our long experience with C has shown such usage to be a rich source of pitfalls. It is much better to use tightly-defined procedures and use statement definitions to achieve the desired higher-level syntax.

## 4 GPSS/H/SLX

SLX's kernel and extensibility mechanisms have been used to create an SLX-hosted implementation of a subset of GPSS/H. Excellent results have been obtained. A large-scale effort to complete the re-hosting of GPSS/H is underway.

The GPSS/H subset is implemented by means of an *include* file, named GPSSH.SLX, which contains definitions of SLX objects which implement GPSS/H entities, e.g., Facilities, Queues, Storages, etc.; definitions of procedures which implement GPSS/H blocks and control statements; and SLX statement definitions which expand into procedure calls and object declarations.

As you might expect, the implementation of GPSS/H exploits features of SLX in a number of interesting ways. For example, consider the problem of producing GPSS/H standard output, i.e., producing a report which shows all Facilities, Queues, Storages, etc. used in a model. In GPSS/H, this is relatively straightforward, since entities are allocated in fixed pools. Standard output is produced by a collection of loops, each of which cycles through all the entities in a given class and prints a report for each entity actually used. In SLX, GPSS/H-style entities can occur as global variables, they can occur as local variables in an unlimited number of widely separated procedures, they can be grouped into arrays, and they can be created and destroyed. Just keeping track of where they all *are* is a difficult problem.

A number of SLX features have been used to solve this problem:

(1) A container object for each entity class is placed into a set called *system*.
(2) The *initial* property for an entity places it into a set contained within its entity class container.
(3) *Report* properties are defined for each entity class.
(4) The report property for a set issues a report for each element in a set.

To produce the equivalent of GPSS/H standard output, one needs only to use a report(system) statement. This issues a report for each entity in each entity class container in the system set. Each entity which exists at the time the report is issued is reported.

For users writing at the GPSS/H level, all of this is accomplished *automatically*. Such users needn't be aware of implementation details. However, users who wish to extend GPSS/H by adding a new entity class of their own design, can easily exploit the mechanisms described above. SLX has a very open architecture.

To test the efficacy of these approaches, we conducted an experiment to see how long it would take to add GPSS/H Logic Switches to SLX. Strictly speaking, Logic Switches are a bit superfluous in SLX, since control variables can accomplish the same (and more) functionality. Nevertheless, it was an interesting experiment. It took us about 30 minutes to add Logic Switches to GPSS/H in SLX. This illustrates the astonishing leverage that can be obtained with SLX.

## 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. The efficacy of the approaches offered by SLX has been demonstrated by re-hosting GPSS/H under SLX.

## REFERENCES

Banks, J., J.S. Carson II, and J.N. Sy. 1989. *Getting started with GPSS/H*. Annandale, VA: Wolverine Software Corporation.

Henriksen, J.O. 1993. SLX: The successor to GPSS/H. In *Proceedings of the 1993 Winter Simulation Conference*, ed. G.W. Evans, M. Mollaghasemi, E. C. Russell, and W.E. Biles. 263-268. Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

Henriksen, J.O., and R.C. Crain. 1989. *GPSS/H reference manual*, third edition. Annandale, VA: Wolverine Software Corporation.

Schriber, T. J. 1991. *An introduction to simulation using GPSS/H*. New York: John Wiley & Sons.

Smith, D.S., D.T. Brunner, and R. C. Crain. 1992. Building a simulator with GPSS/H. In *Proceedings of the 1992 Winter Simulation Conference*, ed. J.J. Swain, D. Goldsman, R. C. Crain, and J.R. Wilson. 357-360. Institute of Electrical and Electronics Engineers, Arlington, VA.

## 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.