Getting Started With SLX

Wolverine Software Corporation
3131 Mount Vernon Avenue
Alexandria, VA 22305-2640
E-mail: Mail@WolverineSoftware.com

1.0 Introduction

This memo presents a sequence of eight case studies which introduce some of the fundamental concepts of SLX. All eight case studies model variations of a system of one or more networked laser printers, starting with a one-line, single-server model, and adding additional complexities which illustrate SLX's modeling capabilities.

2.0 The structure of an SLX Program

An SLX program is a collection of one or more files. Case studies 1A…1H are all single-file programs. Larger models usually make use of *import* statements, which specify the incorporation into higher-level files of definitions contained in lower-level files. SLX's import statement is only superficially similar to C's *#include* statement. While C's #include is a simple device for inserting source code from one file into another, SLX's import statement is considerably more powerful.

An SLX file is a collection of one or more modules. Case studies 1A…1H are all single-module files. SLX modules are the basic encapsulation (packaging) mechanism of SLX. If you prepare a module for use by others, or to serve as your own "toolbox," you can control which elements of the module are private to the module, and which elements are visible to the outside world. Variables can be declared as read-only, allowing a user to access, but not modify them.

Advanced programs may exploit the capabilities afforded by SLX's *precursor* modules. Precursor modules are modules which are *immediately* processed in their entirety by the SLX compiler. When the compiler encounters a precursor module, it sets aside the current program it is compiling (if any) and compiles the precursor module, before resuming its original compilation. Definitions contained in the precursor module are thus available during the remainder of the compilation. This simple device allows you to write executable SLX code which participates in the compilation of a program. SLX itself makes extensive use of precursor modules. For example, data structures such as SLX files, which would otherwise have to be "hard-wired" into the SLX compiler, are defined in precursor modules. This technique allows us to write significant portions of SLX in SLX.

A module contains definitions of global variables which are available throughout the module, and may be made available outside the module by declaring them to be *public.* A module contains definitions of SLX objects (covered below) and SLX procedures. The first several case studies employ a handful of global variables and a single procedure, the mandatory procedure *main.*

3.0 Distinguishing Characteristics of Simulation Languages

There are four essential characteristics of simulation languages which distinguish them from other types of computer languages. Every simulation language must, at a very minimum, include (1) a time-flow mechanism, (2) a means of easily describing activities which happen in parallel, (3) a way of suspending and resuming parts of a model which represent the activities of a system, and (4) means of introducing random inputs and observing, collecting, and analyzing random outputs. In the sections which follow, SLX's implementations of these characteristics are presented.

3.1 SLX's Time-Flow Mechanism

The time-flow mechanism used by SLX is a traditional discrete event mechanism; i.e., time is viewed as a sequence of instants. In any given instant, SLX's simulation supervisor, hereafter referred to as the simulator, processes all the events which can take place at that instant. Frequently, the occurrence of one event in a given instant will trigger another event. For example, when a server completes the service for one of its consumers, and there are other would-be consumers waiting in a queue, an end-of-service event for one consumer will cause a start-of-service event

for another consumer in the same instant of time. When no more events can take place at a given instant, the simulator clock is updated to the next higher time at which an event can take place. Time is a dimensionless quantity; it is the user's responsibility to determine, and use consistently, appropriate time units. In many models, one tick of the clock will be used to represent a second, a minute, or an hour. However, in a model of a computer system, one tick of the clock might represent a microsecond, and in a model of geological phenomena, one tick might represent a year. We must be careful to distinguish the difference between simulated time (in the system being modeled) and real time (the time that it takes the SLX program to run). For example, we will occasionally use the phrase "zero simulated time." This phrase is used to describe activities or changes in the state which take place instantaneously in the system being modeled. Of course, the SLX simulator cannot execute such changes in instantaneous real time.

3.2 Parallelism and Pucks

In SLX, parallel activities are described by using *pucks*, the most important element of SLX. What is a puck? Think of a simulation program as a large sheet of ice. Since a simulation program is used to represent a real system, the program must have parts (areas of the sheet of ice) which correspond to components of the real system. Furthermore, since components of a real system can operate in parallel, a simulation program must create the appearance that different parts of the program are being executed simultaneously. Since SLX can only execute one stream of computer machine instructions at a time, it needs to rapidly switch from simulated component to simulated component in a way which creates the *illusion* of parallelism. For each parallel activity in a simulation, SLX uses a puck to keep track of that activity. The SLX simulator can be viewed as a manager of pucks, moving pucks through the program (around the sheet of ice), and switching from puck to puck as necessary.

The most important property of a puck is its name. SLX puck names are of the form "ssssss i/j". "ssssss" is the name of a component of the program and "i" is a number identifying the particular instance of that component. In case studies 1A…1G, "ssssss i" will always be "main 1", since these case studies have only a main program, and there can only be one instance of a main program. In case study 1H, SLX objects are introduced, allowing names other than main to be used and instance identifiers greater than 1. "j" is a number which identifies each puck associated with a given "ssssss i".

The second most important property of a puck is the location at which the puck will start or resume its motion through the simulation program. We will use the word "poised" to describe this property; e.g., "a puck is *poised* to execute a certain statement in a program." Additional puck properties include its priority, its mark time (the time at which it was created), and its move time (the simulated time at which it will next be eligible to move through the model).

Pucks are created in three ways. "Main 1/1" (the puck for an SLX main program) is automatically created at time zero, with priority -1 (more on that later), poised to execute the first line of the main program. Additional pucks are created by means of the *fork* statement (used in all eight case studies) and by means of the *activate* statement (used and described in case study 1H). The fork statement has the following form:

```
fork
        {
        statements to be executed by the new puck
        }
parent
        {
        statements to be executed by the parent puck
        }
```

When the fork statement is executed, a new puck is created, poised to execute the first statement in the group of statements which immediately follows the fork statement. The original, so-called parent puck continues executing the group of statements in the parent clause. Subsequently (in real time), but in the same instant of simulated time, the new puck will be given its turn to execute. If it completes all the statements in the group following the fork statement, it will skip over the group of statements in the parent clause and continue its execution with the statement following the parent clause. The parent clause is optional; if none is specified, both the parent puck and its offspring puck (if it survives) will execute the next sequential statement.

Pucks are destroyed by executing a *terminate* statement;

3.3 Suspension & Resumption of Puck Movement

As they make their way through a model, pucks can be suspended or forced to wait in three ways: they can wait for scheduled time delays to elapse; they can wait for one or more components of the system to reach required states; and they can be put to sleep (in indefinite waits), to be awakened at some later time by another puck.

Scheduled delays are specified by use of the *advance* statement. The advance statement has the following form:

```
advance expression;
```

When the advance statement is executed, the expression is evaluated and added to the current value of the simulator clock to compute a new move time for the puck which executes the statement. Negative values are not allowed; the clock cannot run backward. Zero values are treated as no-ops. Non-zero values cause a puck to be removed from the list of pucks which are currently eligible to move through the model and to be placed into a list of scheduled future events. For historical purposes, these lists are known as the Current Events Chain and the Future Events Chain, respectively.

State-based delays are specified by using *control variables* and *wait until* statements. Consider the following example, taken from case 1A:

```
control boolean printer_busy;
…
wait until (not printer_busy);
```

The prefix "control" tells the SLX compiler that the boolean (TRUE/FALSE) variable "printer_busy" is a control variable. When a puck executes the wait until statement, if the value of printer_busy is TRUE, making the expression "NOT printer_busy" FALSE, the puck is suspended. The puck is placed on a *reactivation list* associated with printer_busy. At each point in a model where the value of a control variable is changed, the SLX compiler inserts a check to see if any pucks are waiting for that variable to attain a certain value; i.e., it checks for a non-empty reactivation list. All pucks on the reactivation list are reactivated and given another chance to reevaluate their wait until expressions.

Wait until is the fundamental mechanism for synchronizing puck flow in SLX. A wide variety of modeling approaches can be built on top of wait until. This feature is simple but powerful.

3.4 Random Inputs & Outputs

Simulation software must be able to easily incorporate random inputs and to deal with random outputs. To be absolutely proper, we should refer to random inputs and outputs as *pseudo*-random, since random inputs are generated by deterministic algorithms. (If we run the same program twice, without making any changes, identical sequences of random inputs and identical results will be produced.) In other words, the "random" inputs only appear to be random. SLX has an extensive collection of built-in features for generating random inputs and observing, collecting, and analyzing random outputs. These features are described in a companion memo, "SLX Statistical Features."

4.0 Case Studies 1A…1H

4.1 Case Study 1A

Case study 1A is a model of a single, networked laser printer. Jobs to be printed come across the network at a rate of one every 10 to 20 minutes, uniformly distributed. Each job prints for anywhere from 30 seconds to 15 minutes, uniformly distributed. Interarrival times and printing times are generated using the rv_uniform function. This function takes three arguments: the name of a random number stream, a minimum value, and a maximum value. The rn_stream statement is used to define independent streams for generating interarrival times and printing times.

The program is structured as a while loop which executes until the simulator clock reaches or exceeds the value of shutdown_time, which is set to five 8-hour shifts (in minutes). Each circuit of the while loop represents the arrival of a job. The time between jobs is modeled using an advance statement. Following the advance statement, the puck main 1/1 executes a fork statement. The first time around the loop, the fork will create main 1/2, the second time, main 1/3, etc. The offspring puck executes a wait until statement, waiting, if necessary, for the printer to become

available (for the control variable printer_busy to become FALSE). When printer_busy becomes FALSE, the offspring puck sets it to TRUE and executes an advance statement, corresponding to printing time. When the advance time has elapsed, the offspring puck resets printer_busy to FALSE. Note that while the offspring puck is executing the statements in the group following the fork statement, the parent puck (main 1/1) continues independently on its path around the while loop. When an offspring puck is waiting for its printing time to elapse, main 1/1 will be waiting for the time until the next printing job arrival to elapse. If main 1/1 completes its advance before the currently printing puck, the newly arriving puck (forked by main 1/1) will have to wait for the printer.

When an offspring puck finishes printing, it increments a count of the number of jobs printed and terminates. If another job (puck) arrives into the system at a time at which the printer is still printing a prior job, the new puck is suspended at the wait until statement. When the current job resets printer_busy to FALSE, any jobs (pucks) which arrived after it will re-execute their wait untils, and the first such puck will find that the printer is now available, and be allowed to print. Any others in line behind the first one will find that printer_busy has once again been set to TRUE.

Once the shutdown time has been reached or exceeded, main 1/1 prints the number of jobs printed and shuts down the run. The model is shut down summarily; i.e., a printing job (if any) is not allowed to complete, and any jobs waiting for the printer are ignored. Output is produced using the *print* statement. The print statement specifies a list of variables to be printed and a picture into which the values are to be edited. Underscore characters indicate fields in the picture into which values are to be edited. Vertical bars within or at the end of a group of underscore characters indicate how the edited data is to be justified:

|_____	left-justified item
_____|	right-justified item
____|____	centered item

If an edited value is too big to fit within the specified field, it is squeezed in; i.e., the remainder of the picture is pushed to the right.

4.2 Case Study 1B

Case study 1B adds more realistic shutdown conditions and basic statistics to Case study 1A. To facilitate more realistic shutdown conditions, two counters are added, jobs_in, and jobs_printed. Both are defined as control integer variables, allowing their use in wait until statements. Jobs_in and jobs_printed are incremented when a new job arrives and when a job completes printing, respectively. Upon escaping from the while loop, main 1/1 executes the following wait until:

```
wait until (jobs_printed == jobs_in);        // drain the system
```

If there is a job printing and if there are jobs in the queue, jobs_printed will be less than jobs_in. Jobs_printed will equal jobs_in only after the last job has printed. Of course, throughout the simulation, there will be many times at which jobs_printed will equal jobs_in. However, each time either variable changes, the simulator will find that no pucks are waiting for changes in value. Only after main 1/1 escapes the while loop does the detection of value changes become significant.

Case study 1B produces statistics for the average time in queue and for the utilization of the printer. Average time in queue is computed as the total time pucks have spent in queue, divided by the number of jobs printed. Average utilization of the printer is computed as the total time the printer was active, divided by the total run time. The accumulation of total time in queue and total time the printer was active requires a little explanation. The time in queue for any given job is the time it starts printing, minus the time that it came into the system. This expression must be computed in the instant at which the job starts printing. At that time, the current value of the simulator clock is available in a simulator-maintained variable named **time.** A job's time of entry into the system is available in the mark_time property of the puck, but we haven't yet seen how to access puck properties. To get around this, we do the subtraction of arrival times from the total queueing time at the time of arrival, and we add the time at which printing starts when the job starts printing. In other words instead of specifying

```
total_queueing_time += time - "arrival time";
```

We do it in two steps:

```
total_queueing_time -= time;      (executed at arrival time)
total_queueing_time += time;      (executed when the job starts printing)
```

The computation of printer utilization is done using the same technique. Note that use of this technique requires that the system not be shut down until the queue is drained and the server is idle. If we fail to shut down properly, we run the risk that the subtraction step above may have been done for one or more jobs, but the addition step may not yet have been done. The longer the model runs, the larger the value of **time** becomes, and the greater the error.

4.3 Case Study 1C

Case study 1C explores what would happen if instead of having one laser printer, we had two laser printers which operated at half the speed of the original printer. What would you expect to happen? (Consider the limiting case where every job has its own printer. What happens to queueing times? What happens to printer utilization and "time through the system" performance measures?)

In Case study 1C, the boolean control variable printer_busy is replaced by a control integer variable, jobs_printing. Each time a job starts printing, jobs_printing is incremented, and each time a job finishes printing, jobs_printing is decremented. The wait until condition used to model availability of a printer becomes

```
wait until (jobs_printing < 2);
```

4.4 Case Study 1D

Case study 1D adds more detail to the modeling of the arrival of jobs to be printed. In this case study, three different streams of jobs are modeled, one for managers, one for workers, and one for accountants (which are neither managers nor workers). Each job stream has its own distribution of random interarrival times and random printing times. Because all three streams are identical except for their interarrival and printing times, a single procedure (subroutine), named job_stream, has been added. This procedure is parameterized to accept the characteristics of each job stream. The while loop of case studies 1A…1C has been moved out of the main program and into the job_stream procedure. The main program executes three forks, creating pucks main 1/2, main 1/3, and main 1/4. Main 1/2 models manager jobs; main 1/3 models worker jobs; and main 1/4 models accountant jobs. Once these pucks have been created, at any given point in time, we have three pucks executing the while loop inside job_stream.

Main 1/1 waits for shutdown as follows:

```
wait until (time >= shutdown_time and jobs_printed == jobs_in);
```

This is an example of a compound wait until, i.e., a wait until which evaluates multiple conditions. When **time** is used in a wait until expression, it must be used alone; i.e., "wait until time + 10.0 > 20.0" is not allowed. Other than this, there are no limitations on the complexity of wait until expressions.

4.5 Case Study 1E

Case study 1E is identical to case 1D, except that the three fork statements in the main program have been replaced by a single fork statement inside the job_stream procedure. This raises an interesting point. When a fork is executed inside a subroutine, which pucks are eligible to return to the procedure which called the subroutine? They *all* are. Use counts are maintained for data which is local to a procedure. Each time a new puck is created, the use count for the data of the procedure containing the fork is incremented. The use count is decremented when the new puck terminates or returns from the called procedure to the calling procedure. The data which is local to a procedure is released only when its use count goes to zero.

In this case study, three independent instances of job_stream are active. Each instance has its own copy of data local to job_stream. In this simple example, the only variables which are local to job_stream are its arguments. This is, of course, exactly what we want to achieve. We want to model three independent streams of job arrivals, each with its own unique characteristics. The use of multiple instances of a single procedure allows us to avoid using three separate procedures. Each offspring puck created by the fork statement inside job_stream shares its parent's job_stream data.

What are the puck names for the three instances of job_stream? They're still main1/2, main 1/3, and main 1/4. Calling a procedure does not cause a new puck to be created.

4.6 Case Study 1F

Case study 1F is identical to Case study 1E, except that the fork statements for individual job_streams are moved back into the main program, and priorities are specified for them. By default, pucks created by fork statements take the priority of their parent; however this can be overridden by using a *priority* clause. The main procedure operates at priority -1. In larger models, pucks other than main 1/1 typically run at priorities greater than or equal to zero. Following this convention assures that all model activities are allowed to complete prior to main 1/1's completion of a time advance (corresponding to the planned duration of a run) or successful completion of a wait until (state-based model termination).

4.7 Case Study 1G

Case study 1G presents a world view which differs from that used in case studies 1A…1F. The prior case studies have all used an "active object, passive server" world view. In each of the models we've seen up to this point, jobs have been modeled as active entities (pucks) which flow through a system, competing for servers which are inherently passive. We've used a control boolean variable to represent a single server and a control integer to represent multiple, identical servers. In many modeling situations, this simplified view of servers is entirely adequate. However, in many, if not most, real-world systems servers exhibit complex behavior which is anything but passive. Consider, for example, modeling a butcher in a grocery store. In a very simple model, we could represent the butcher as a passive server, using the techniques presented in case studies 1A…1F. In a real grocery store, a butcher exhibits complex, active behavior. For example, if he's not busy, he takes time to rearrange meat in the display cases. Several times a day, he has to go to the back room and cut more meat. If the store has a deli department, he may have to prepare meat for their use. And the list goes on. To model such a server, we need to use *executable* logic. In SLX, this implies that the butcher would have to be represented by a puck.

Case study 1G runs with four pucks, main 1/1 and its three offspring. Main 1/1 is used to represent the printer (in a single printer system), and its offspring are used to model the arrivals of manager, worker, and accountant printing jobs. The model is written as a producer-consumer model. The three pucks representing arrival streams produce jobs which are consumed by the printer. Printer jobs are represented as objects created from an SLX object class named printer_job. Printer_job objects are passive; i.e., they have no direct association with any puck. They are created by means of the *new* operator:

```
pointer(printer_job)     job;
…
job = new printer_job;
```

Job is a pointer variable which is defined as pointing to printer_job objects. Each printer_job object has three elements: job_priority, job_arrival_time, and job_printing_time. Once values for these object elements have been set, the object is placed into an SLX set named printer_queue. The printer_queue set is ranked in order of descending priority.

Main 1/1 calls a procedure named printer. Inside the printer procedure, main 1/1 executes a *forever* loop. At the top of this loop, it waits until the size of the printer_queue set is greater than zero. When this condition is true, main 1/1 removes the first job from the printer queue, processes it in accordance with the characteristics deposited in the object when it was created, and destroys the object. At the bottom of the loop, main 1/1 tests whether the shutdown time of the system has been reached and the printer queue is empty. If so, main 1/1 returns from the printer procedure to the main program, prints summary statistics and terminates the program.

For a system as simple as our networked laser printer, the approach described above may seem to be a bit of overkill; however, in more complex models this approach is very powerful.

4.8 Case Study 1H

In case studies 1A…1F, we used an active object, passive server modeling world view. In case study 1G, we showed how to implement active servers. Many components of systems are neither purely active nor passive. Such components may act upon other components or be acted upon by other components during various stages of their lifetimes. Case study 1H introduces the concept of the active object. Active objects can both act and be acted upon.

In case study 1H, the printer procedure of case study 1G has been replaced by an active object named printer. An active object is distinguished from a passive object by the presence of an *actions* property. An active object must be created by using the new operator and activated using the activate operator. Usually these two steps are combined:

```
activate new printer();      // start an active object
```

In case study 1H, main 1/1 executes the above statement, and waits until the control variable shutdown becomes TRUE. The activate statement creates a new puck for an object. In this case the puck created will be printer 1/1 (the first puck for the first (and only) instance of a printer object). When the new puck is created, it is poised to execute the first statement of the object's actions property. Superficially, the use of an active object in case study 1H looks very similar to the ways in which we have used procedure in prior case studies. Let's explore the differences.

When a procedure is invoked, the path back to its caller is preserved. If a called procedure calls even lower level procedures, the entire call path must be preserved. Preservation of the call path is very complex. Traditional programming languages use a stack architecture for this purpose. For each procedure call, new space is allocated on the stack (usually by decrementing a stack pointer) for storing the path back to the caller, arguments (if any) to the called procedure, and data local to the procedure. In SLX, life is complicated by the fact that forks can take place (necessitating the use of use counts, as described above), and the fact that advance statements or wait untils can cause execution of the currently active puck to be suspended. Each puck must have its own call stack, portions of which may be shared with other pucks.

In contrast to procedure calls, after an active object is activated, all implicit connections to the puck which created it are severed. An object can have an *initial* property, which is executed under the auspices of the puck which issues the "activate" statement, but after the initial property is executed, there is no direct connection to the activating puck. Up until now, we have regarded a simulation program as a single sheet of ice, over which the simulator moves pucks. Activating objects creates new sheets of ice. We can now envision a model as a collection of parallel planes.

Active objects have all of the characteristics of passive objects. They can have arbitrarily complex elements; they can be placed in sets; pointers can be used to refer to them, etc. They also have all of the executable characteristics we have seen in case studies 1A…1G. They can fork; they can wait until; they can advance; and they can terminate. Active objects are the workhorses of SLX.

5.0 Conclusions

In the foregoing sequence of case studies, we have examined most of the fundamental characteristics of SLX's approach to modeling. The set of SLX kernel simulation primitives is both remarkably small and remarkably powerful. If you examine these case studies to the point of truly understanding them, you will have mastered most of the simulation architecture of SLX. Of course, this is only part of the picture. On top of these simulation primitives, you have a procedural language which implements a fair subset of C. Perhaps most importantly, the SLX compiler includes extremely powerful capabilities for extending the base language to handle your needs in ways that you desire. When you're first learning SLX, it's important that you understand how things work at the kernel level. Once you've begun to tailor SLX to your needs, you'll spend most of your time working at higher levels of abstraction. If you're modeling transportation systems, your SLX programs may look quite different from those of someone modeling telecommunication systems. Think of SLX as an inverted, truncated pyramid. The collection of kernel primitive to be mastered has a small "footprint," but the collection of applications for which it is an appropriate modeling tool is vast.