A 30-Minute Guided Tour of SLX

1 Introduction

This memo provides a 30-minute guided tour of SLX, illustrating the basic structure of an SLX program and the architecture of SLX. The tour uses SLX's interactive debugging tools to present key elements of a simple queueing model. The system modeled is a single chair barbershop, the first model of Thomas J. Schriber's legendary *Simulation Using GPSS* textbook.

This memo is intended for readers who are familiar with at least one traditional simulation language, e.g., GPSS/H™, Siman™/Arena™, Slam™/AweSim™, Simscript II.5™, etc. Throughout the tour, we'll relate SLX's terminology and features to those commonly found in traditional simulation languages.

2 Getting Started

Let's get started! You can use either SLX Professional or Student SLX for this tour. The installation procedure for either version of SLX will create a Wolverine folder on your desktop, containing one or more icons for invoking SLX.

When you invoke SLX, you should expand the SLX window to full-screen size, since the tour will display many small windows, taking up a significant portion of the screen. The initial SLX main window comprises a menu bar, a toolbar, a "Log" window, a "Trees & Expansions" window, and a status bar at the bottom of the screen. The Log window will be used for displaying program output and SLX debugger information as we work our way through the barbershop model. The Trees & Expansions window will provide visual roadmap of where we are in the SLX program.

Now let's open the barbershop model source file, barb137.slx. To do so, click on File, Open. Next, select barb137.slx from the SLX Sample folder. At this point, the text of the barbershop model will be placed into the large, formerly open portion of the SLX window.

3 Compiling the Program

To compile the program, click on the "check-mark" icon in the toolbar. The icons in the SLX toolbar use Windows tooltips, so if you don't know or can't remember the meanings of the toolbar icons, you can just hold the mouse pointer over them, and tooltip reminders will be displayed. Note that if SLX does not currently have input focus, Windows will not display tooltips. (SLX should lose focus only if you switch out of the SLX window to do something else.)

When you compile the program, SLX will reformat the source code window, using colors and boldface text to enhance the readability of the source program. The following visual cues are provided:

A. "Ordinary" program source code is shown in dark blue.

B. Comments are rendered in a gray color. SLX uses C/C++ notation for comments. In lines containing "//", all text from the "//" to the end of the line is treated as a comment. Alternatively, "/*" and "*/" are used as start-of-comment and end-of-comment delimiters, with everything in between treated as a comment.

C. Reserved SLX keywords, such as **import** and **module** are shown in dark blue boldface. Such keywords are hard-wired into the syntax of SLX and cannot be used as variable names in your program.

D. SLX defined statement names, such as **facility** and **rn_stream**, are shown in bright blue boldface. Defined statements are macro-like extensions to the basic SLX language. These extensions are expanded into lower-level SLX statements. We'll take a detailed look at several of them later. Note that were it not for the bright blue coloring, defined statements would otherwise be indistinguishable from built-in SLX statements. The ability to preserve the look and feel of SLX in user- or system-defined language extensions is a hallmark of SLX

E. Built-in function names, such as rv_uniform (for generating uniformly distributed random samples), are shown in a purple color. To see rv_uniform, you'll have to scroll down the source text a bit, since it's below the initially visible text.

F. If you encounter any errors, or if warning messages are issued, messages are rendered in red, and source text is highlighted in red.

Take a look at the Log window. You'll see a summary of the compilation of the barbershop model. Note that a lot of underlying code (not shown in the source window) is compiled, so that the total number of lines compiled is substantially larger than the number of lines in our sample model. Take a look at the effective lines per second speed of the SLX compiler shown in the Log window. Just for fun, click the check-mark icon to recompile the program. Second and succeeding compilations are always faster than the first, because the program source text is retained in memory. On fast machines, it's not unusual to see compilation rates in excess of 400,000 lines per second. Thus, a 5,000-line program is child's play for SLX.

4 The Structure of an SLX Program

4.1 Source Files

An SLX program is comprised of one or more source files. Our barbershop model contains an "**import** <h7>" statement, which imports a file named h7.slx. The ".slx" suffix is assumed. H7.slx is a file developed by Wolverine Software that implements many of the commonly used higher-level building blocks of GPSS/H. The notation "<h7>" tells SLX that this file is contained the SLX system directory. Imported files names that are enclosed in quotation marks, e.g.,

‘**import** “myfile.slx”’ are taken from your working directory, which will ordinarily be different from the SLX system directory.

Import statements must be placed at the top of a source file, before any non-comment statements. Imported files are processed in *depth-first* order. For example, the file h7.slx includes an “**import** <stats>” statement, which imports lower-level definitions for statistics collection required by h7.slx. The depth-first approach implies that stats.slx will be compiled first, h7.slx second, and barb137.slx, our sample program, last. The depth-first approach guarantees that lower-level definitions are processed before they are used at higher levels. SLX automatically handles redundant **import**s; i.e., if an **import** statement is encountered for a file that has already been processed, it will be ignored, rather than giving rise to a large number of “duplicate definition” messages. (Those of you who have programmed in C/C++ will appreciate this. Managing multiple #include statements in C/C++ is painful.)

4.2 SLX Modules

Each SLX source file contains one or more modules. Modules provide convenient packaging features. For example, we can declare a module to be “public read_only”. Variables defined in such a module are (by overridable default) visible in other modules of an SLX program, but within the other modules, the variables can only be read, not written. If you are building software to be used by others, and you want to prevent inadvertent modification of variables that only your code should be allowed to modify, the packaging capabilities of modules can be very helpful. Even if you’re writing code only for your own personal use, it often makes sense to use such techniques to enforce a discipline of access to variables to avoid shooting yourself in the foot.

Like C/C++, SLX uses braces (“{ }”) to delimit the scope of major syntactic units. Thus, the module named barb13, containing our barbershop model, extends from the “{“ following the name barb13, through the final “}” at the bottom of the program.

Variables that are defined at the module level are global variables. Subject to restrictions such as public, private, read_only, write_only, etc., they can be accessed throughout an SLX program. Our sample program defines five global variables, joe, joeq, Arrivals, Service, and stop_time.

“joe” is defined as a GPSS/H facility. In GPSS/H, facilities are single server entities. In our single-chair barbershop, joe is a passive server used to model the barber. By *passive* server, we mean that joe only responds to requests for service made by active entities (customers). Joe has no active behavior pattern of his own.

“joeq” is a queue in which customers wait for service by joe. In GPSS/H, the use of queues is optional; queues are provided only for the purpose of collecting statistics on queueing. In GPSS/H, a model runs identically whether or not queues are used to collect statistics. In other languages, e.g. Siman/Arena, queue entities provide somewhat different functionality. We’ll discuss how queues work in GPSS/H in greater detail below.

The barb13 module contains two global random number streams, named Arrivals and Service. These random number streams are used for generating customers' random interarrival and service times, respectively.

Finally, barb13 contains a global variable named stop_time, which is used as the time at which shutdown of the barbershop is initiated. Stop_time is defined as a constant, which means it can't be modified, and assigned an initial value of 480, which corresponds to one 8-hour day, expressed in minutes.

Like C/C++, SLX is a strongly-typed language requiring that variables be defined before reference.

4.3 SLX Classes

Let's continue our examination of the structure of the barbershop model. Following the definitions of the barb13 module's global variables is a definition of a *class* of "customer" objects. This class defines the properties and behavior of the customers that flow through the barbershop. A typical SLX model will contain many class definitions. Note that the body of the customer class is enclosed within braces "{ }". Section 5 presents SLX class concepts in detail.

4.4 SLX Procedures

The final major component of our barbershop model is the main procedure. (In SLX, the main program, subroutines, and functions are called procedures.) Because the barbershop model is very simple, its comprises only a single procedure. In a larger model, a multiplicity of procedures would be used. The role of the main procedure is described in Section 6. Note that the body of the main procedure is enclosed within braces "{ }".

5 SLX CLASSES

5.1 Object-Based Modeling

To build a model using any simulation tool, we must examine a "real" system and decide which components of the real system are important enough to require their inclusion in the model, and for components included in the model, we must decide which of their attributes are important enough to also warrant inclusion. Rarely can we include all components and all their attributes. The essence of the modeling process is developing an *abstract* representation of a system. The abstract representation must have sufficient fidelity to the real system to produce meaningful results. However, if the level of detail is allowed to become too great, the likelihood of producing meaningful results in an acceptable time frame decreases.

SLX *objects* are used to represent components of a system. Every object is an *instance* of a *class* of objects. A class is a generic pattern, and objects are individual instances of that pattern. It may be helpful to think of a class as a cookie cutter used to cut out individual cookies (objects). In our barbershop model, we define a single class, customer, that defines the properties and behavior of

a generic customer. At any given point in a simulation, we may have zero, one, or more customer objects in the barbershop.

Our model makes use of several predefined classes of objects. For example, our barber, joe, is a *facility* (single server), and facilities are defined as an object class in h7.slx. Likewise, the *queue* in front of joe, joeq, is defined as an object class in h7.slx.

5.2 Object Attributes

Attributes of an object are defined as variables within the object's class. SLX allows the use of integer, floating point, string, boolean, pointer, and enumerated type variables as object attributes, as well as *sets* of objects. Enumerated type variables are variables that can be assigned only values from a list of names. For example, a variable used to represent a machine's state could be represented as an enumerated type with three named values: "idle," "running," and "broken." Sets of objects can also be used as object attributes. For example, a server object might include a set of request objects.

5.3 Active and Passive Objects

SLX has two kinds of objects, active and passive. Passive objects are objects that can only be acted upon. Active objects have dynamic behavior patterns that describe the rules by which they operate. The behavior pattern of an active object is specified in an *actions* property in its class definition. (Those of you who are familiar with object-oriented languages can regard SLX *properties* as standard SLX methods, some of which have built-in defaults.) In our barbershop model, our customer class has an actions property. Take a quick look at it. (We'll step through it in detail in Section 7.) Note that the actions property is enclosed in braces "{ }", as is always the case with SLX's major syntactic units.

When we build a model, we must decide which objects are active and which are passive. In our barbershop model, customers are modeled as active objects, and joe the barber is modeled as a passive object. This is a very natural way to describe the operation of a barbershop, because our attention naturally focuses on the flow of customers from the customers' perspective. Poor joe simply stands around for eight hours, reacting to requests for service. If our model were a bit more realistic, joe might have a behavior pattern that included such activities as eating lunch and going to the bathroom. In a more realistic model, we would almost certainly model joe as an active object.

The "active consumer, passive server" world-view was first popularized by GPSS in the early 1960s. This world-view has been adopted by most discrete event simulation tools. While SLX supports this world-view, it by no means forces it upon you. It would be possible to write an SLX barbershop model in which customers were modeled as passive objects, and the barber was modeled as an active object. To do so, we'd probably introduce a fictitious arrivals process, modeled as an active object, to bring passive customer objects into the model.

Finally, note that in our simple model, no attributes are really required for customer objects. We have, however, included a "customer_dummy" integer variable, to show you where attribute definitions would be placed.

6 The Main Procedure

In SLX, the main procedure provides overall control of a simulation run. Our main procedure does three things: (1) it defines the rules by which customer objects arrive at the barbershop; (2) it waits for the completion of a day's worth of operation; and (3) it requests printing a report of summary statistics for the operation of the system. In Section 7, as we step through the model, we'll discuss these three functions in detail.

A typical SLX main procedure would contain local variables used to implement a collection of experiments to be performed with the model. Our barbershop model is very simple. It runs for one day and prints results. Accordingly, there is no real need for local variables in our main procedure; however, we have included a variable named "main_dummy," just to show you where definitions of local variables would be placed.

7 Stepping Through the Barbershop

7.1 Getting Started

If you haven't already done so, bring up the barb137.slx model, and click on the check-mark icon to compile the model. If you've scrolled down the model source code, please scroll back up to the top.

7.2 Debugger Windows

Click on Monitor (near the center of the main window menu bar), and then click on Basic Debugging Information. Three windows will appear, a Local Data window, a Global Data window, and an "All Pucks" window. The source code window will shrink to accommodate the debugger windows. SLX always places source code windows in the largest rectangle remaining after non-source windows have been placed. Non-source windows can be placed and sized automatically, as you have just seen, or manually. In either case, SLX will adjust the size and placement of any source code windows.

7.2.1 The Global Data Window

Maximize the Global Data window (located in the lower left corner of the screen) by clicking on the square icon in its title bar. The Global data window shows all module-level variables of all modules comprising a program. As you can see, there are many global variables in addition to the aforementioned five defined in the barb13 module. By default, variables are sorted in ascending alphabetical order. Click on the "+Variable" column header. The header will change to "-Variable", and variables in the window will be sorted in reverse alphabetical order. Click one more time, and the header will change to "Variable", and variables will be displayed in the order

in which SLX encountered their definitions. As a consequence of SLX's depth-first approach to compilation, lowest-level definitions will appear at the top of the list, and variables defined in the barb13 module will appear at the bottom.

Next, click on the "Module" column header. The column header will change to "+Module", and variables will be sorted in ascending module name order, and ascending variable name order within modules. You should now see at the top of the Global Data window the five global variables defined in barb13. Click once more on the header. The header will change to "-Module", and reverse alphabetical order will be used.

Click on the "Type" column header to sort global variables by type. Click again to sort by type in reverse order. Finally, click once more on the "Module" column header to force our barb13 global variables to the top of the window.

The "Value" column shows the values of scalar variables. For global variables that are more complex, e.g., sets, arrays, or objects, additional windows must be activated to display their contents, since they don't have a simple, single value. We'll illustrate how to do this later. Please reduce the currently maximized Global Data window to its normal size by clicking on the double square icon at the right end of the toolbar before continuing.

7.2.3 The Local Data window

The Local Data window displays variables local to the current procedure or object. Since we haven't begun executing the model, the Local Data window is currently empty. Click once on the "Step Over" icon (a yellow box with an arrow above it), or press the F10 key. SLX supports four forms of stepping through a program, step over, step into, step out of, and step-by-count. For the time being, we'll use only the step over form.

As a consequence of clicking on the "Step Over" icon, the main procedure has now been activated, and the **arrivals** statement (the next statement to be executed) is highlighted in red. Local variables of the main procedure now appear in the Local Data window. The "Variable" and "Type" column headers can be used for altering the sorting of variables in this window, in the same manner we saw earlier in the Global Data window.

Two variables appear in the Local Data window, main_dummy, which we discussed above, and #Uses. The local (*instance*) data for every SLX procedure and object has an SLX-maintained use count. The use count represents the number of ways in which the local data can be accessed. This count plays an important role when an attempt is made to destroy an object and when return is made from a procedure. In both cases, if the use count for the local data is greater than zero, the local data cannot be destroyed, because there are still ways in which the data can be accessed within the program. Conversely, if the use count of a procedure's or object's local data goes to zero, the local data is *automatically* released, because all possible ways of accessing the data have been eliminated. SLX's use counts prevent "gone, but not forgotten" and "forgotten, but not gone" memory management errors. The use count of the main procedure's local data is 1. We'll explore this later.

7.2.4 The All Pucks Window

Real systems include interactions among system components that operate in parallel. To properly mimic a real system, a simulation model must appear to be able to carry on multiple activities simultaneously, with proper synchronization, over *simulated* time. In reality, a computer program running on a single computer can do only one thing at a time, but by rapidly switching from activity to activity in accordance with well-designed rules, a simulation program can create the *illusion* of being able to do many things at once. In discrete event simulation, simulated time is viewed as a succession of *instants*. At any given instant in simulated time, a simulation language's run-time executive program, which we'll call "the simulator," must process all events that can take place at that time. Having done so, the simulator must then advance the simulator clock to the next imminent event time. For each ongoing activity, the simulator needs to keep track of at least three things: (1) the location of the next statement to be executed for the activity in the program, (2) the location of data local to that activity, and (3) the simulated time at which the activity is to resume (when activities undergo scheduled time delays). In SLX, pucks are used to keep track of this information.

Why did we choose the name puck? (You won't find this nomenclature in any other simulation tool.) Think of a simulation program as a sort of hockey game with a set of rules or flowchart written on the ice. The logic by which each system component operates occupies a portion of the ice. To keep track of where it is in the logic of each active component, SLX uses pucks. Thus, the sheet of ice typically has many pucks at any given point in simulated time, one for each ongoing activity. At each instant of simulated time, SLX moves all the pucks ahead in their logic as far as they can be moved. The forward movement of a puck is stopped for one of five reasons: (1) the puck incurs a scheduled time delay, such as the duration of a period of service; (2) the puck must wait for one or more system components to attain a given state or states, e.g., wait for a server to become free; (3) the puck deliberately places itself into a "sleeping" state, to be awakened later by another puck; (4) the puck is destroyed, representing the cessation of an activity; or (5) the puck causes a fatal execution error.

Each puck has a name and two ID numbers. In the All Pucks window, you can currently see the puck "main 1/1." "main" is the name of the currently executing procedure. The "1" to the left of the "/" identifies the particular instance of main. Since we can never have more than one main program, the instance ID for main will always be 1. Later we'll see the objects "customer 1," "customer 2," etc. created in sequence. The number to the right of the "/" identifies the particular puck associated with the instance of procedure main or instance of an object. In general, it's possible to have more than one puck associated with a given procedure/object instance. We'll see how this works momentarily.

The "T' column in the All Pucks window contains a "T" for pucks whose trace flag has been turned on. When a puck's trace flag is turned on, each time the simulator is about to try to move the puck forward in the model, execution is stopped, enabling you to carefully follow the puck's progress.

The “Move Time” column shows the time at which a scheduled time delay will complete, or the time at which the most recent such delay was completed.

The “Priority” column shows a puck’s priority. SLX always attempts to move pucks in order of descending priority.

The “Puck State” column shows the current state of a puck. In the current display, the state of puck main 1/1 is “moving.” This tells us that this is the puck that the simulator is currently attempting to move forward in the model.

7.3 The Barbershop Model Under the Microscope

Click once on the “Step Over” icon (or press the F10 key) to begin executing the main procedure. The **arrivals** statement will be highlighted in red, indicating that it is the next statement to be executed. The **arrivals** statement is an extension to SLX defined in h7.slx. Right click on the **arrivals** keyword. A popup menu will appear, presenting you with a number of options for getting more information about the **arrivals** statement. Click on “Show arrivals’s Expansion.” The source code window will be expanded to show the lower-level SLX statements that the **arrivals** extension expands into.

The first statement of the **arrivals** statement’s expansion is a *fork* statement. To see what this statement does, click on the “Step Over” icon once more. Two things will happen. First, the puck main 1/1 that we’re following will move ahead to the **wait until** statement following the expansion of the **arrivals** statement. Second, a new puck, main 1/2 will appear in the All Pucks window. We now have two pucks, main 1/1 and main 1/2, for the main procedure. This enables the main procedure to “do two things at once” in *simulated* time. Note that puck main 1/1 is still in a “moving” state, while puck main 1/2 is in a “movable” state, which indicates that it is eligible to execute at the current simulation time. Main 1/2 will gain control of the simulation after main 1/1 has completed its actions at the current time (zero).

Creation of a second puck for the main procedure has increased the use count of main’s local data from 1 to 2, indicating that main’s data can now be accessed in two ways. Verify this by examining “#Uses” in the Local Data window. We can determine exactly what the two means of access are by right clicking on #Uses and clicking on “Show main’s Uses (2)” in the popup menu that appears. Please do so, and a “Uses of Main” window will be created. Within that window, we see that the main 1/1 and main 1/2 pucks both contain references to main’s local data. The notation “*puck_object” means that there is a *pointer variable* named puck_object that points to the data in question. In the current model, the connections between main’s data and its pucks are pretty obvious; however, in a larger, more complex model, such relationships might not be so obvious. For example, if we’re trying to destroy an object that is no longer in use, but the object’s use count is greater than 1, we may need to track down the outstanding ways in which the object can still be accessed. Right clicking on #Uses makes this easy. To conserve screen space, please kill the “Uses of Main” window before proceeding.

Left click on the name portion of the main 1/2 puck in the All Pucks window. The "{" inside the expansion of the **arrivals** statement will be highlighted in red. This tells us that when the simulator attempts to move the main 1/2 puck, the body of the fork will be executed. Left click on the name portion of the main 1/1 puck in the All Pucks window. The **wait until** statement that main 1/1 is about to execute will once again be highlighted in red.

The **wait until** statement says that main 1/1 must wait until three conditions are simultaneously true: (1) **time**, the value of the simulator's clock, must reach or exceed stop_time, the constant with a value of 480 we defined above; (2) the queue named joeq must be empty; and (3) the facility named joe must not be in use. The value of "stop_time" can be observed by holding the mouse over the variable name. The latter two conditions are specified using built-in the macros Q(queue name) and FNU(facility name), respectively. Holding the mouse over the macro names causes explanations of their functionality to pop up.

Variables used in a **wait until** statement must be declared to be *control* variables. When a variable is defined as a control variable, each time its value is changed, SLX tests to see if any pucks are waiting in **wait until** statements for a change in the variable's value, and if so, the puck is given a chance to re-test its **wait until** condition. The size of a queue and the busy/not busy state of a facility are defined as control variables. The simulator clock ("**time**") is treated as a control variable, but SLX is able to optimize the handling of **wait until** conditions including **time**-based expressions. We can translate the **wait until** statement into English as follows. At the end of an 8-hour day, wait until no customers are in line waiting for a haircut, and no customer is currently getting a haircut. In other words, wait for the barbershop to be empty and idle. This example illustrates the utility of *compound* **wait until** statements. This **wait until** has one time-based condition and two state-based conditions combined in a single, easily readable SLX statement.

Click on the "Step Over" icon to execute the **wait until** statement. Two things will have happened: (1) in the log window, we are told that puck main 1/1 is now waiting; and (2) in the All Pucks window, the state of main 1/1 has been updated to "scheduled." You might wonder why the state is not "waiting," rather than "scheduled." This is because SLX is very smart about how it processes **wait until** conditions. It knows that if the first of three conditions is false in an expression where all three conditions must be simultaneously true, there's no point in even looking at the latter two conditions. Thus, as a first step in executing the **wait until**, SLX schedules a time delay, preventing retesting of the compound condition until time 480. This is a very efficient way to handle this kind of waiting.

Take a look at the All Pucks window, and place yourself in the "shoes" of the simulator. There are two pucks in the system. One is in a scheduled state, and the other is in a movable state. The only rational action at this point is to try to move the movable puck. To verify the simulator's actions, click on the "Pick up the next puck that can move" icon in the toolbar. (This icon contains a green-colored left bracket ("["). The Log window will indicate that puck main 1/2 is now moving, and the state of main 1/2 in the All Pucks window will be updated to "moving."

The first statement to be executed by main 1/2 is a **forever** statement. This statement is an intentionally infinite loop; i.e., the statements within the braces ("{ }") that follow the **forever** keyword are to be executed repeatedly. Exit from the loop, if any, can occur only as a consequence of statements within the body of the loop. Click on the "Step Over" icon until the **advance** statement is highlighted in red. The **advance** statement is used for scheduled time delays. In the current **advance** statement, the time delay to be scheduled is random, uniformly distributed between 12 and 24, i.e. 18±6. The randomly generated times in the current **advance** statement represent the times between successive arrivals of customer arrivals into the model. Since no previous customers have arrived, we're currently generating the time of the first customer arrival. Click on the "Step Over" icon to execute the **advance** statement. Two things will happen: (1) the Log window will tell us that puck main 1/2 is advancing to time 15.243546 (a random value), and (2) the state of main 1/2 in the All Pucks window will change to "scheduled."

Once again, place yourself in the shoes of the SLX simulator. There are two pucks in the system; both are in a scheduled state; and there are no more moving or movable pucks at the current instant (time zero) in simulated time. Therefore, your next action is to update the simulator clock to the time of the next imminent event. Main 1/1 is scheduled to move at time 480, and main 1/2 is scheduled to move at time 15.243546. Therefore, the clock must be updated to this time. Click on the "pick up the next puck" icon to verify that this is what happens. You'll see three things happen: (1) the Log window will indicate that puck main 1/2 is now the active puck; (2) the state of puck 1/2 will be updated to "moving;" and (3) the status bar at the bottom of the screen will be updated to show the current state of the model.

Main 1/2 will next test whether the simulator clock (**time**) has reached or exceeds stop_time, and if so, puck main 1/2 will be terminated (exit the model). Click on the "Step Over" icon to verify that termination is skipped over because **time** is 15.243546, which is less than the stop_time value of 480.

At this point, the "**activate new** customer" statement will be highlighted in red. Click on the "Step Over" icon to see what happens when this statement is executed. Two things have happened: (1) a new puck, customer 1/1 has appeared in the All Pucks window, in a "movable" state; and (2) the main 1/2 puck we've been tracking has reached the end of the **forever** loop. Click the "step over icon" until the advance statement is reached at the top of the loop. The verb **new** creates a new instance of a class of objects, and the verb **activate** creates a puck for the new object instance. Click on the name "customer" of the customer 1/1 puck in the All Pucks window. You will see that the next statement this puck will execute is the first statement contained within the **actions** property of the class customer.

At this point, let's re-examine the **arrivals** statement. This statement specifies three things: (1) the class of object for which arrivals are to be scheduled, (2) the time between successive arrivals, usually referred to as interarrival time, or IAT, and (3) the time at which the arrivals process is to be terminated. The **arrivals** statement can be viewed as a kind of high-powered macro whose arguments ("customer," "rv_uniform(Arrivals, 12.0, 24.0)," and "stop_time") are inserted into lower-level SLX statements resulting from expansion of the macro.

It's interesting to note that the **arrivals** statement appears to be a *declarative* statement. Only by examining its expansion, and the **fork** statement in particular, do we see that the **arrivals** statement is implemented as a very interesting sequence of *executable* statements. We've explored the details of the **arrivals** statement to further our understanding of SLX, but to use this statement in the future, we'd probably be willing to treat it as a declaration and ignore its inner workings. There are many feature in SLX that share this characteristic. You can use them as is, but if you're really curious, you can explore their inner workings. SLX has a very open architecture.

Let's resume execution of the barbershop model. Click on the "Step Over" icon. Main 1/2's execution of the **advance** statement has placed it in a "scheduled" state. At this point, customer 1/1 is movable, and both main 1/1 and main 1/2 are scheduled. Since the simulator has a movable puck at the current instant of time, it will attempt to move it forward in the model. Click on the "Pick up the next puck" icon to verify this. You will see that customer 1/1 is now poised to execute an **enqueue** statement. The **enqueue** statement will record the time at which customer 1/1 entered the queue named joeq. Click on the "Step Over" icon to execute the **enqueue** statement.

Customer 1/1 is now poised to execute a **seize** statement. The **seize** statement will attempt to acquire the facility (single server) named joe. We know that this attempt will succeed, because we are tracking the first customer of the day, so no other customer can currently possess joe. For the sake of argument, suppose we didn't know this. Right mouse click on the name joe in the **seize** statement. (An alternative method for getting details about joe is to right click on joe in the Global Data window. At present, it's easier to click on joe in the **seize** statement.) A popup menu will appear, offering you a number of options for obtaining further information about joe. Click on "Show joe's Report." This will open a window showing GPSS/H standard statistical output for facility joe. Note that joe's "Seizing Puck" is currently <NULL>. Click on the "Step Over" icon to execute the **seize** statement. Joe's Seizing Item will become customer 1/1.

Customer 1/1 is now poised to execute the **depart** statement, which will remove it from the queue named joeq, recording the time customer 1/1 spent in the queue (in this case no time at all). Click on the "Step Over" icon, executing the **depart** statement. Right mouse click on joeq, and select "Show joeq's Report" from the popup menu that appears. Take a quick look at the statistics that have been observed so far for joeq. Since we won't be looking at joeq for quite a while, kill the "joeq Report" window.

Customer 1/1 is poised to execute an **advance** statement that will delay its progress for a random time, uniformly distributed between 12 and 18 (15±3). This time delay represents the duration of customer 1/1's haircut. Click on the "Step Over" icon to execute the **advance** statement. Carefully examine the All Pucks window. You will see that all three pucks are in a scheduled state, and there are no moving or movable pucks. The next imminent event time is 33.1546, the move time for customer 1/1. Main 2, which is currently undergoing the time delay between the arrival of customer 1 and customer 2, is scheduled to move at time 33.2358. This tells us that

customer 1 will exit the model before customer 2 arrives. Therefore, customer 2 will not have to wait for joe.

Click on the "Pick up the next puck" icon. Customer 1/1 is now poised to execute the **release** statement, which will free the facility named joe. Click on the "Step Over" icon to execute the **release** statement. Note that "Seizing Item" in the joeq Report window has reverted to <NULL>. Customer 1/1 is now poised to execute the **terminate** statement, which will destroy the customer 1/1 puck. Click on the "Step Over" icon to execute the **terminate** statement. The customer 1/1 puck has now disappeared from the All Pucks window, and the Local Data window has been emptied, because there is no currently active puck.

Examine the All Pucks window. Both pucks are in a scheduled state. Therefore, SLX will update the simulator clock to time 33.2358, the next imminent event time. Click on the "Pick up the next puck" icon and note that the current time has been updated in the All Pucks window title bar and in the status window at the bottom of the screen. Click on the "Step Over" icon to execute the current **if** statement. Since we haven't yet reached time 480 (stop_time), the **terminate** statement will not be executed. Click on the "Step Over" icon, executing the "**activate new** customer" statement. Puck customer 2/1 will appear in the All Pucks window, in a "movable" state. Main 1/2 will have reached the end of the **forever** loop and have wrapped around to its **advance** statement.

Click on the "Pick up the next puck" icon. Doing so will allow main 2 to be scheduled and will cause customer 2/1 to be picked up and changed from a "movable" state to a "moving" state. Step through customer 2/1's **enqueue**, **seize**, **depart**, and **advance** statements. At this point, you will see three scheduled pucks in the All Pucks window. SLX will update the simulator clock to 47.9093, customer 2/1's move time. Click on the "Pick up the next puck" icon to verify this, and then step through customer 2/1's **release** and **terminate** statements.

Click on the "Pick up the next puck" icon to pick up main 1/2. Click on the "Run until the current puck stops moving icon" to move main 1/2 as far as it can go. We haven't used this icon yet. It includes a red-colored right bracket ("]") and is located just to the right of the "Pick up the next puck" icon. Verify that main 1/2 has gone around its **forever** loop, created and activated customer 3/1 and has been delayed until time 68.5622.

Click on the "Pick up the next puck" icon, picking up customer 3/1. Since facility joe is currently not in use, we know that the **seize** statement will succeed, and customer 3/1 will proceed without delay to its **advance** statement. Click on the "Run until the current puck stops moving" icon.

Carefully examine the All Pucks window. You will see that all three pucks are in a scheduled state. Note, however, that main 1/2 is scheduled to move at time 68.5622, *before* customer 3/1, which is scheduled to move at time 70.1637. Thus we can see that customer 4/1, which will be created by main 1/2, will arrive before customer 3/1 finishes its use of facility joe, and, customer 4/1 will experience a non-zero delay waiting for joe to be free.

Pick up the next puck (main 1/2). Move it as far as it can go. (Remember the "Move the current puck as far as it can go" icon?) At this point, you should drag the bottom border of the All Pucks window down a bit to clearly show all four pucks in the window. Note that customer 4/1 is the only movable puck. Pick it up and step through its **enqueue** and **seize** statements. Note that execution of the **seize** statement causes customer 4/1's state to transition from "moving" to "waiting," and a message stating that "customer 4/1 is now waiting" has been written in the Log window.

Note that the customer 4/1 is now waiting in a **wait** list statement, highlighted in red. We can explore the complete context of customer 4/1 by examining the Calls & Expansions window. This window shows that customer 4/1 has executed a seize statement that has in turned called a procedure named SEIZE. Click on Customer 4/1 in the Calls & Expansions window. Doing so will bring the main source window back to the top, and the color of customer 4/1's SEIZE call (within the expansion of the **seize** statement) is now green. The green color indicates that customer 4/1 is currently "in" a statement at a lower level of SLX than that shown in the current source code display. Clicking on SEIZE in the Calls & Expansions window takes us back to the currently executing statement in the window containing h7.slx. We're not going to go into further details here, but suffice it to say, customer 4/1 has placed itself into a list of pucks waiting for a change in facility joe's seizability, and it has "gone to sleep." Prince Charming is on the way, however.

Once again, click on customer 4/1 in the Calls & Expansions window. This will get us back up to the source code of our barbershop model. Take a look at the All Pucks window. You will see that there are three scheduled pucks and one waiting puck (customer 4/1). Click on the puck name for each puck in succession to verify where the puck will next execute. The next puck that will move is customer 3/1. Pick up customer 3/1. It is poised to execute its **release** statement, which will free facility joe.

At this point we're going to step into the logic of the **release** statement. Click on the "Step Into" icon (a yellow box with an arrow that extends to the right and down). Alternatively, press the F11 key. This will take us into customer 3/1's execution of the RELEASE procedure, which implements the **release** statement. The first statement to be executed in RELEASE is an **if** statement that verifies that the puck releasing a facility is the same puck that previously seized it. Step ahead to execute the **if** statement. Next, you'll see logic for handling facilities that have been preempted. Since our model does not use preemption, this logic will be skipped over. Execute the **for** statement. Customer 4/1 is now poised to execute a **tabulate** statement, which is an SLX extension statement for collecting statistics. Step *into* the **tabulate** statement (two clicks). Drag the bottom border of the Calls & Expansions window downward far enough to make its entire tree visible. Step through a few statements of the slx_tabulate_observation procedure. You'll find them largely incomprehensible. There are two points to be made here, however. First, you'll rarely find it necessary to delve into the execution of SLX-hosted GPSS/H statements at this level of detail. GPSS/H is, after all, a high-level language. Second, if you ever *do* need to have access to low-level execution details, that access is available. SLX has a *very* open architecture. Only SLX kernel-level details are inaccessible.

Let's work our way back to more comfortable ground. Click on the "Step Out Of" icon (a yellow box with an arrow going up and to the right). Alternatively, press the F9 key.This will take us out of slx_tabulate_observation and back to the RELEASE procedure. The Calls & Expansions window will be updated to reflect our return from slx_tabulate_ob-servation and completion of the **tabulate** statement. Step through the next several statements until you reach the first **reactivate** list statement. Step once more to execute this statement, and observe very carefully what happens. This statement reactivates any pucks that are waiting for a a change in the facility's ownership. (Note that it is possible to wait for such conditions without competing to seize the facility.) Since there are no pucks waiting for such a condition, nothing happens. Step one more time to execute the **if** statement. The **if** statement tests whether facility joe is currently "available." This terminology is a bit unfortunate. Availability is not the same as seizability. GPSS/H has statements for making a facility available or unavailable. These statements can be used for such things as making a worker unavailable for use during his/her lunch hour or for making a machine unavailable for use during periods of breakdown. The relationship between availability and seizability is further complicated by GPSS/H options such as allowing a puck that has seized a facility to complete its usage of the facility even though the facility has been made unavailable. This option could be used for modeling a checkout clerk's taking a break in a supermarket. When the clerk's scheduled break time arrives, he/she does not suspend checking out his/her current customer. Rather, he/she places a "closed" sign behind the last customer in his/her queue. Our barbershop model does not make use of facility availability/unavailability, so facility joe is always available. Execute the **if** statement. Customer 3/1 is now poised to execute a **reactivate** statement that will reactivate all pucks (if any) that are waiting to **seize** joe. Watching the All Pucks window very carefully, execute the **reactivate** statement. As a consequence of customer 3/1's execution of the **reactivate** statement, customer 4/1's state has changed from "waiting" to "movable." Execute customer 3/1's **return** statement; click once more; and we'll be back in the source code of our barbershop model. Execute customer 3/1's **terminate** statement. Step once more and verify that customer 3/1 disappears from the All Pucks window.

At this point, there is one movable puck (customer 4/1) and there are two scheduled pucks (main 1/1 and main 1/2). Pick up the next puck (customer 4/1). Note that customer 4/1's move time is now in the past, indicating that it tried to move earlier but was delayed. Move customer 4/1 as far as it can go at the current time. Look at the All Pucks window. You'll see three scheduled pucks, the first of which is main 1/2. Because main 1/2 has a lower move time than customer 4/1, customer 5/1, which main 1/2 will create, will arrive before customer 4/1 completes its use of joe. Customer 5/1 will incur a non-zero delay waiting for joe.

Perhaps by now, you've had enough. Let's do one more thing before we quit. Right click on the name "main" for puck main 1/1 in the All Pucks window. Click on "Set Puck Trap" in the popup menu that appears. This will stop execution when main 1/1 tries to move at time 480. Click on the "running man" icon to run the model to time 480. Note that facility joe is currently in use by customer 26/1. Execute main 1/1's **wait until** statement. Main 1/1 will transition from a "moving" state to a "waiting" state, because facility joe is still in use, and therefore main 1/1's **wait until** expression is still false.

Pick up the next puck, main 1/2, and execute its **if** statement. Since we have reached time 480, the **if** condition is true, and main 1/2 will **terminate**. Execute main 1/2's **terminate**. This will destroy main 1/2, and since main 1/2 was the puck that created and activated arriving customers inside the expansion of the **arrivals** statement, no more customers will arrive. Relief is in sight for joe. Pick up the next puck, customer 26/1, and execute its **release** and **terminate** statements. At this point, the only puck left in the All Pucks window is main 1/1. As a consequence of customer 26/1's releasing facility joe, main 1/1 is once again in a "movable" state. Pick it up, and execute its **wait until** statement. This time (time 488.0056), the **wait until** condition will be true, and main 1/1 will fall through to the "**report** system" statement. Execute this statement, and statistics for one day's operation of the barbershop will be written to the Log window. Maximize the Log window if you want to look examine these statistics. This concludes our guided tour of the barbershop.

8.0 SUMMARY

Let's review what we've seen in our guided tour. We've seen that the structure of an SLX program is a collection of files and modules within files. We've reviewed some of the packaging features afforded by using modules. We've seen the central role played by objects in building SLX models, and we've seen two kinds of objects, active and passive. In our barbershop model, we used two kinds of predefined passive objects, facilities and queues, and we defined an active object, customer, of our own.

We've explored in great detail the relationship between SLX pucks and active objects, and we've used the SLX debugger to examine exactly how pucks are processed when scheduled and state-conditioned delays occur. We've seen two kinds of parallelism. First we saw how the **fork** statement can be used to create additional pucks that allow a single object or procedure to "do more than one thing at once." The **arrivals** statement exploited this feature for scheduling arriving customers. The second form of parallelism we saw was interaction among different objects. We saw a puck delayed because the puck for another object had seized the facility joe that both pucks wanted. The second form of parallelism, interaction *among* objects, can be found in virtually all discrete-event simulation tools. The first form of parallelism, multiple simultaneous activities *within* an object, is unique to SLX.

We've seen a little bit of SLX's extensibility. Extensibility allows us to add higher level statements to the SLX language. These statements expand into lower level statements. All the extensions we looked at were prepackaged for us in h7.slx; however, the tools used to build the extensions have been designed for use by end users of SLX. We like to say that extensibility is the tool of second resort in SLX. The tools of first resort are SLX's built-in capabilities. When no tool is available to exactly meet your needs, you can build your own tools, using SLX's extensibility features. With SLX, you'll never get stuck. You'll always be able to program your way out of situations where you need additional capabilities.

If you want to learn SLX, you have to bring your brain with you. SLX is not a tool for dummies. SLX *forces* you to come to grips with basic simulation concepts. If you don't understand the difference between a scheduled time delay and a state-conditioned delay, for example, you'll be

hard pressed to understand the roles of **advance** and **wait until**. Conversely, mastery of **advance** and **wait until** brings you up to speed on two fundamental simulation concepts that can be applied repeatedly. All-told, the number of such SLX kernel simulation primitives is surprisingly small (about a dozen). Thus, mastery of a relatively small number of concepts empowers you to tackle simulation projects of unlimited complexity.

Finally, through the use of the SLX debugger, we've seen first-hand SLX's open architecture. We've done our best to hide details you don't need or want to know without making them inaccessible. Being hidden and being inaccessible are the same only when a "black box" approach is taken. Most simulation software uses the black box approach. If you don't know how a given feature works, you have to *read* about it or *call* the vendor for technical support. With SLX, and the SLX debugger in particular, you have the opportunity to *explore* the software's inner workings and *see* for yourself exactly how things work. You are empowered.

SLX is a very powerful simulation tool. It has powerful and innovative features, many of which can be found in no other software.