A Comparison of SLX and GPSS/H
For Experienced GPSS/H Users

1. Introduction

In developing SLX, we wanted to preserve the conceptual strengths of GPSS/H, while going well beyond it in terms of extensibility, modularity, and generality. If you know GPSS/H, virtually all of your knowledge about simulation and modeling will be transferable to SLX, but since GPSS/H concepts have been implemented within a new and much more general framework, you'll have to shift gears a bit to get into SLX. To help you get started, this memo describes some of the major differences between SLX and GPSS/H.

2. The Layered Architecture

SLX was explicitly designed to support hierarchical modeling. We wanted users to be able to build constructs that operate at a higher level than GPSS/H blocks, and then to be able to (re)use those constructs as components in still higher-level constructs, and so on. We also wanted users to be able to develop new constructs at approximately the same level as GPSS/H blocks, and – where desirable – at lower levels, too.

GPSS/H is essentially a single-layer software package. You can extend it to two layers by using external routines written in C or Fortran, but doing that requires going *outside* of GPSS/H itself. The internals of GPSS/H blocks are hidden from you, and only Wolverine Software can add new blocks (or control statements). MACROs provide a way of "packaging" linear sequences of GPSS/H blocks and/or control statements for reuse, but the MACRO mechanism was never designed to provide the kind of heavy-duty nested aggregation that hierarchical modeling requires.

To address the needs of hierarchical modeling, SLX was conceived from the beginning as having multiple layers. Whether such an architecture will be successful depends on three properties. First, the individual layers must be well conceived. Our twenty years' experience designing, implementing, and working with GPSS/H provided plenty of insight into what the layers should contain. Second, the layers must not be too far apart, so that shifting one's attention up or down from one layer to the next is a smooth transition, not a jarring loss of focus. Third, there must be good mechanisms for getting from one layer to the next. If you're working at a given layer, it should be relatively easy for you to add to that layer by using components taken from lower layers. For example, if none of the prepackaged GPSS/H-style "blocks" provided with SLX meets your needs, you can build your own "block" *using the same lower-level components and tools that we use.* It should also be easy to build higher levels on top of your current level.

SLX provides three mechanisms for extensibility: objects, macros, and user-defined statements. A detailed explanation of these mechanisms is beyond the scope of this memo; however, a brief summary follows. SLX is an *object-based* language. It is not truly *object-oriented*, because it lacks inheritance and procedural polymorphism which are key components of the OOP paradigm. SLX utilizes a *composition* paradigm for construction of complex objects; i.e., large objects are built by incorporating smaller objects into their definitions. Using this bottom-up approach, one can construct objects of arbitrary complexity. For example, if you wanted to construct a "super-Facility," you could define an object which incorporated a GPSS/H Facility and additional properties of your own.

SLX has extremely powerful macro definition capabilities. SLX's statement definition capability is an extension of the macro facility which allows you to add your own statements to SLX. For both macros and statement definitions, you supply a prototype which defines the components and syntax of the macro/statement, and you supply a *definition* section which expands the definition into SLX source code. Prototypes accommodate a wide variety of syntactic forms, e.g., positional parameters, keyword parameters, optional parameters, punctuation characters, variable-length arguments lists, etc. Any language which includes macros must include a *macro language* for defining the rules by which a macro is expanded. In most languages, the macro language is distinctly different from the host language. Macro languages typically offer much weaker expressiveness than their host language. For example, #if, #else, and #endif in the C/C++ preprocessor macro language are much weaker than the conditional branching and looping language constructs available in the C/C++ host language. In SLX, the macro language is SLX itself; i.e., the full range of SLX's expressive power can be used in macro/statement definitions. The code that you provide in macro/statement definitions is pre-compiled by SLX into machine instructions which execute as an extension of the

SLX compiler. In other words, you can write code which runs at *compile* time. This approach affords virtually unprecedented expressive power. An example which illustrates this power is the collection of statements we defined for using Proof Animation in conjunction with SLX. We defined a "layout" statement which reads a Proof layout (.lay) file at compile time and constructs a compile-time dictionary of Proof path and object class names. We defined a collection of statements for writing commands to a Proof trace (.atf) file. For each command implemented, in contexts where a path or object class name must be specified, the dictionary can be used to validate the existence of the component during compilation of the SLX model. In other words, the SLX-Proof interface "knows" about the contents of a layout.

3. Source Code Appearance

The format of GPSS/H statements resembles that of an assembly language (label, verb, comma-separated operands). Statements must be in upper case, and operands cannot contain imbedded blanks. By contrast, SLX statements are completely free-format and can use both lower- and upper-case letters. SLX's syntax is very similar to that of C. If you're familiar with C, the transition from GPSS/H to SLX will be easy for you. If not, you should find the syntax reasonably intuitive, but you'll have to learn to remember to put semicolons at the ends of statements.

4. Integer Indices

In GPSS/H, all entity classes (Facilities, Storages, Queues, etc.) are stored in pools whose sizes are fixed once a model begins execution. Ultimately, every access of a GPSS/H entity is expressed using an integer index into the pool for the given entity type. If symbolic names are used (SEIZE JOE), the names are converted by GPSS/H into integer indices. A user can define the symbol-to-integer mapping in an EQU statement when desired, but otherwise GPSS/H will automatically establish a mapping. When contiguous groups of entities are used in a model in combination with indirect addressing, incorrect computation of integer indices can have serious consequences. For example, assume that in a model of a computer network Facilities 1-10 are used to represent CPUs, and Facilities 11-20 are used to represent disks. If a model erroneously attempts to SEIZE CPU number 11, disk number 1 will be SEIZEd instead, but no error will be detected because 11 is a valid Facility number. Nor is the error easily detected by the modeler, since one small integer (10) looks pretty much like another (11).

In SLX, on the other hand, instances of each entity type are no longer stored in a single pool. A modeler can explicitly declare an array of objects, and each such array is stored (and addressed) separately. In the computer network example, we would define two separate arrays of Facilities, each of size 10. An attempt to SEIZE CPU 11 would result in a clearly identified run-time error, since there are only 10 CPUs in the array. It's even possible to dynamically create and destroy GPSS/H entities during execution of a model. This makes it easy to build generic models in which a wide variety of model configurations can be created in response to user inputs or specifications.

5. GPSS/H Transactions vs. SLX Active Objects

The original version of GPSS was designed by Geoffrey Gordon in the early 1960s. He *invented* the Transaction-flow world view. His landmark achievement made it possible for "non-programmers" to easily describe processes that operate in parallel. (Even in the 1960s, "programming vs. non-programming was a hot issue.) Although GPSS has undergone many changes over the years, Transaction-flow is still the heart of the language.

In GPSS/H, a Transaction has two internal parts. The first is a "header", which contains all of the information necessary to support a Transaction's movement through a model. Included in the header are the transaction's priority, a pointer to the next GPSS/H Block it is to execute, internal chain pointers, etc. The second part of a Transaction is its "parameter area", which contains all user-defined Transaction data. The numbers and kinds of Parameters are specified in the GENERATE or SPLIT Block which creates the Transaction.

In SLX, the role of the GPSS/H Transaction has been split into two pieces. User-defined data consists of attributes defined for an *active object.* These attributes are not limited to integer and floating point values as is the case in GPSS/H. Strings, boolean values, enumerated types can be used. Furthermore, arrays and sub-objects can be used..

The SLX equivalent to a GPSS/H Transaction header is called a *puck*. In order to further discuss the role of the puck, which is central to SLX, we must first examine SLX objects in general.

SLX provides two kinds of objects – active and passive. *Passive objects* are simply instances of named data structures that are acted upon by an SLX program. (They're like C structs.) For example, in a model of a networked printer, we could define a passive object class representing jobs to be processed by the printer as follows (note availability of the enum data type):

```
class printer_job
    {
    enum(normal, legal) paper_type;
    int                 page_count;
    };
```

Active objects have, in addition to any necessary data items, an *actions* property specified in their class definitions. The actions property describes the behavior of the object over its lifetime. The actions property corresponds to a sequence of GPSS/H Blocks following a GENERATE or SPLIT Block. The following example illustrates a laser printer modeled as an active object. In the example, the printer spends its entire lifetime circulating through a forever loop. The printer waits for at least one printer_job object to show up in the input queue, removes the first printer_job from the queue, prints the job represented by the printer_job object (advances by the print time), then destroys the printer_job object. In the advance statement, note that the job object's paper_type, which is of enum type, can be used *directly* as an index into an array of printing speeds which depend on paper_type.

```
class printer()
    {
    pointer(printer_job)  job;

    actions
        {
        forever
            {
            wait until (printer_queue.size > 0);

            job = first printer_job in printer_queue;
            remove job from printer_queue;

            advance (job->job_page_count * time_per_page[job->paper_type]);

            destroy job;
            }
        }
    };
```

SLX objects – both passive and active – are dynamically created using the *new* operator. Pointer variables are used for keeping track of dynamically created objects:

```
pointer(printer_job)  p;

p = new printer_job;
p->paper_type = legal;
p->page_count = 27;
```

```
place p into printer_queue;
```

Active objects must be *activated* once they have been created. The *activate* verb creates a puck for the just-created printer object and then places the puck on the Current Events Chain, poised to execute the first statement of the printer's actions property. Pucks are the schedulable entities of SLX. Pucks are placed on the Current Events Chain or Future Events Chain, and when blocking conditions occur in a model, pucks wait for the blockages to be removed. Creation and activation of an active object can be accomplished by:

```
pointer(printer)        ptr;

ptr = new printer;
activate ptr;
```

The above sequence can be abbreviated as:

```
activate new printer;
```

It is beyond the scope of this memo to describe all the actions pucks can perform or have performed on them. With a rough idea of the role of SLX pucks in mind, however, we can resume our comparison of GPSS/H and SLX.

A GPSS/H Transaction may be thought of as combining an SLX active object and a puck into a single, indivisible unit. In GPSS/H, all scheduling operations are performed using Transactions. Transactions are placed on the Current and Future Events Chains, and Transactions wait for blocking conditions to be removed. Transactions all have identical formats; they differ only in their numbers and kinds of numeric parameters (fullword, halfword, byte, and floating point). If we were to describe GPSS/H Transactions as SLX active objects, all Transactions could be described as a single object class, treating parameters as numeric arrays with variable dimensions. In contrast, SLX active objects have unlimited variety..As we'll see in the next section, multiple operations can be scheduled simultaneously for any given active object; i.e., an active object can have more than one puck. In short, the expressive power of SLX objects is far greater than that of GPSS/H Transactions.

SLX's "activate new" most closely resembles GPSS/H's SPLIT block. In GPSS/H, the SPLIT block is executed by a parent Transaction and creates one or more offspring Transactions. The offspring Transactions execute code whose label is given as the B-operand of the SPLIT block. In SLX, the type of object executing an "activate new" may differ from the type of object which is created and activated; hence there is no implicit copying of attributes from parent to offspring. (If desired, such copying must be done explicitly.) The offspring object executes the code defined in its object class's actions property, rather than having its behavior specified by the object responsible for its creation.

6. Describing Parallel Activities

The reason that we use simulation languages, rather than procedural languages, for writing discrete event simulation models is that simulation languages provide easy-to-use, built-in features for describing parallel activities. One measure of the quality of simulation software is the ease with which this can be accomplished. In GPSS/H, parallelism always takes the form of interactions among Transactions. For example, many Transactions can be in ADVANCE blocks at the same time. Similarly, some Transactions can be in control of Facilities and Storages, while other Transactions must wait before they can SEIZE or ENTER, respectively, those Facilities and Storages.

In SLX, there are two levels at which parallelism can be modeled. Large-scale and small-scale (or local) parallelism are modeled using different mechanisms. *Large-scale parallelism* is modeled as interactions among SLX active objects, and closely resembles the GPSS/H "interacting Transactions" approach. *Local parallelism*, on the other hand, is modeled using the SLX *fork* statement, which creates an additional puck for the current active object. When a fork statement is executed, both the parent and offspring pucks share access to the same active object. For example, suppose that we model a complex machine as an SLX active object. Within the machine, several subsystems may operate in parallel. Other subsystems may remain idle until external sensors are activated or deactivated. By using fork statements to create multiple pucks for a given machine, all of these activities can take place in parallel within the

single SLX object used to model the machine. Each puck has direct, albeit shared, access to the machine's properties. I we have multiple machines, each is characterized by data local to the machine and a family of pucks with shared access to that data.

In GPSS/H there is a two-level hierarchy for sharing model data: all data is either local to a given Transaction, or it is global and accessible to all Transactions. The sharing of data local to Transactions (Transaction Parameters) can require considerable effort in GPSS/H, because Transactions do not have direct access to other Transactions' parameters. If we were to use GPSS/H to model the complex machine from the previous example, we would have to model the parallel activities by SPLITting off a number of offspring Transactions. The data to be shared among those Transactions would have to be global, since placing it in any one of the Transactions would be impractical. If we only had one such machine, we could use a collection of Savevalues. If we had more than one such machine, we would have to use Matrix Savevalues and maintain integer indices into the matrix rows or columns. Alternatively, we could use indirect addressing to access contiguous ranges of (scalar) Savevalues. In either case, considerable effort can be required to accomplish the goal.

In SLX, there is a three-level hierarchy for data sharing. Data can be global, local within procedures (functions and subroutines), or local within objects. Using objects and pointers to them, sharing data is very straightforward. Any object has easy access to all of the local data of other objects (except data that is defined as private to a given object). In addition, as we discussed above, if multiple pucks are created for a given object, they all have direct access to the object's data.

7. Active Objects as Switch Hitters

One of the fundamental decisions that must be made when developing a simulation model is which components need to be modeled as passive components and which need to be modeled as active components. The natural world view of GPSS/H is "passive resource, active consumer." In GPSS/H, consumers of resources are almost always modeled as Transactions, while the resources themselves are modeled as Facilities or Storages which have no active behavior of their own and can only be acted *upon*. When the behavior of a resource is complex, however, it is often necessary to model the resource using a Transaction. For example, consider how a butcher would be represented in a model of a supermarket. In a simple model, the butcher could be modeled as a Facility. Requests for butcher service would take the familiar form of "QUEUE, SEIZE, DEPART, ADVANCE, RELEASE." In a more detailed model, the complexity of a "real" butcher's activities would have to be taken into account. During otherwise idle periods, the butcher must rearrange meat in the display cases, cut new meat, prepare meat for the deli department, and so on. Modeling such activities would almost always necessitate representing the butcher using a Transaction. Unfortunately, a butcher Transaction cannot also function as a butcher Facility, so a Facility and a Transaction would have to be used in combination.

In SLX, active objects can easily assume *passive roles*. This is almost always accomplished by having an active object place its puck into a set and then execute a *wait* statement. A wait statement with no *until* clause is treated as a "put me to sleep" operation. Pucks that execute such unconditional wait statements are said to be in an indefinite wait condition. They wait until they are reactivated (awakened) by another puck that examines the set into which the waiting puck placed itself. Consider the problem of representing a job that is making its way through an assembly process. If the process is complex, we might want to model the job using an active object. But suppose that at certain times the job has to be transported from point A to point B by a conveyor, forklift, or other agent. If the actions of the transporter are complex, then it, too, should be modeled using an active object. During the time that the assembly job is being transported, it can assume a passive role and be manipulated by the transporter object.. After the job arrives at point B, it can resume its active role. This kind of "switch hitting" occurs in many modeling contexts and is handled easily in SLX.

8. The Current Events Chain

Like GPSS/H, SLX has a Current Events Chain (CEC). In SLX, however, the operation of the CEC is much simpler. In GPSS/H, Transactions that are experiencing non-unique blockages (e.g., at refusal-mode TEST blocks) are all retained on the Current Events Chain. Each time that such a Transaction is examined during a CEC scan, the blocking condition is retested. Transactions that are experiencing unique blocking conditions (e.g., waiting to SEIZE a Facility) are also retained on the CEC, but their "scan-skip indicators" are turned on, so they are efficiently skipped over during CEC scans.

In SLX, *all* blocking conditions are unique. SLX's *wait until* statement – a true breakthrough in simulation language design – uses one or more control variables in an expression describing a blocking condition. When a blocking condition exists, so that the expression evaluates to FALSE, the puck that must wait is removed from the CEC and placed into a reactivation list associated with each control variable for which the puck must wait. When the value of a control variable is changed, a test is made to see whether any pucks are waiting for a change in that variable's state. If so, the waiting pucks (kept on the control variable's reactivation list) are reinserted into the CEC and allowed to retest their wait-until expressions. SLX thus combines tremendous ease in specifying arbitrarily complex blocking conditions with optimally efficient determination of when the conditions have been removed.

Consider Tom Schriber's barbershop model, the grandfather of all GPSS models. The first barbershop model that a student develops typically has unrealistic shutdown conditions. At 5:00, the model simply stops, ignoring a haircut in progress and any customers waiting in the queue. In a second model, more realistic shutdown conditions can be implemented. At 5:00, the barbershop door is closed, and the current haircut (if any) is completed, and haircuts are given to customers in the queue (if any) before the model is shut down. SLX and GPSS/H source code for this approach are shown in Appendix A.

9. Practicing Safe Simulation

One of GPSS/H's greatest features is its total run-time error checking. It is impossible to access beyond the end of a MATRIX SAVEVALUE, for example, without getting a run-time error. SLX carries on this tradition. Given the greater computational complexity of SLX, this presented a major challenge in its development, but we think that you'll be pleased with the results. For example, *all* pointer-variable references are validated at run time. For those of you who've ever had to track down a reference to a NULL or uninitialized pointer-variable in a C program, this feature should be greatly appreciated (and a real eye-opener as well).

10. Subroutines, Modules, Files

In GPSS/H, models are most often written as "one big program," contained in a single file. In part, this is because GPSS/H does not support "full-blown" subroutines as they occur in general-purpose programming languages. Although TRANSFER SBR provides a mechanism for calling and returning from subroutines, all arguments to the subroutine must be placed into the parameters of the Transaction that will execute it. Any local data items in the subroutine must also be stored in parameters of the Transaction executing it.

SLX is a marriage of GPSS/H-like simulation capabilities with the rich expressiveness of C. Consequently, SLX supports true subroutines. In addition, SLX incorporates modules, which are collections of subroutines and (data and object) definitions. Modules are the principal packaging mechanism of SLX, and they allow access to the internal details of a module to be carefully controlled. You can make accessible only those features that need to be accessible from outside the module, while hiding the internal implementation detail from the outside world.

11. Conclusions

SLX preserves much of the best of GPSS/H's rich legacy and familiar concepts, but in a new framework which offers much greater power and generality. Although some effort is required to learn the new language, that effort can pay large dividends.

Appendix A — GPSS/H and SLX Barbershop Models

On the next page, source code is shown for a GPSS/H implementation of a barbershop with realistic shutdown conditions. Since this memo is intended for experienced GPSS/H users, we will not discuss the GPSS/H model. On the following page is an SLX implementation of the same model. Salient features of the SLX model will be highlighted in the paragraphs which follow. The first statement of the program, "import <h4>", imports h4.slx, a file which defines an SLX-based dialect of GPSS/H. The SLX model, *per se*, is divided into three sections: global declarations, customer flow, and run control.

In the global declarations section, the Facility and Queue used in the model are declared. In SLX, all variables must be declared prior to their first usage. This is in sharp contrast to GPSS/H, in which variables are implicitly defined by their occurrence in a model. Random number streams used in the model are defined. Note that the specification of random number stream starting positions is optional. If no starting position is specified, SLX will choose the next higher multiple of 100,000 beyond the highest starting position encountered in previously specified streams, if any. The final declaration, stop_time, is provided only for enhanced readability.

The customer flow section of the model comprises an SLX active object class named customer. The sequence of statements is directly analagous to the GPSS/H model, apart from the absence of an SLX equivalent to the GPSS/H GENERATE Block.

The run control portion of the model is provided by the SLX main program. The *arrivals* statement specifies that customer objects are to be created and activated with uniformly distributed interarrival times until the model's stop_time has been reached. The arrivals statement provides the functionality of the GPSS/H GENERATE Block. In GPSS/H, GENERATE Blocks can be used on a free-standing basis. Each GENERATE Block produces a stream of Transactions which flows into the Blocks which follow. In SLX, the arrivals statement acts more like a GPSS/H control statement than a Block. The arrivals statement expands into a fork statement which creates a second puck for the main program. The body of the fork is a loop which is executed until the stop_time is reached or exceeded. Each time through the loop, main's second puck executes an *advance* statement to model the interarrival time between successive customers, and it activates a new customer object, e.g. "activate new customer". Each customer created and activated begins its execution with the first statement of the actions property of the customer object class.

The "report(system)" statement produces standard output for the SLX model. "system" is an SLX set. The members of the system set are entity class container sets. In this model, there is are entity class container sets for facilities, queues, and random_streams. "joe" is the only member of the facility set; and "joeq" is the only mamber of the queue set. "Arrivals" and "Service" belong to the random_stream set. In SLX, an object class can have a *report* property. (This is the second example of an object property; we've already discussed the actions property above.) A report property does the obvious. In h4.slx, we have defined report properties for each GPSS/H entity class. The report statement, e.g., "report(system);" invokes the report property for the specified object or set of objects. The report property for a set invokes the report property for each member of the set (if any). Thus, "report(system)" issues a report for each entity of each GPSS/H entity class. This provides for the SLX equivalent to GPSS/H standard output.

Finally, the SLX model is terminated by an optional C-style "exit(0);" statement. If this statement were not provided, flow of execution off the end of the main program would be interpreted as a normal program exit.

```
       SIMULATE
*
*       ONE-LINE, SINGLE-SERVER QUEUEING MODEL
*
        GENERATE        RVUNI(1,18,6)   ARRIVALS EVERY 18 +- 6 MINUTES
        GATE LR         SHUTDOWN        NOT AFTER SHUTDOWN
        QUEUE           JOEQ            GET IN LINE
        SEIZE           JOE             GRAB THE BARBER
        DEPART          JOEQ            EXIT LINE
        ADVANCE         RVUNI(2,15,3)   HAIRCUT TAKES 15 +- 3 MINUTES
        RELEASE         JOE             FREE THE BARBER
        TERMINATE       0               EXIT THE SHOP
*
*       TIMER SEGMENT
*
        GENERATE        ,,48000,1,-1    RUN FOR 800 HRS (IN MINUTES)
        LOGIC S         SHUTDOWN        ENTER SHUTDOWN MODE
        TEST E          Q$JOEQ,0        DRAIN QUEUE
        GATE NU         JOE             WAIT UNTIL JOE IS IDLE
        TERMINATE       1               SHUT DOWN
        START           1
        END
```

```
//*****************************************************************************
//          SLX Barbershop Model
//*****************************************************************************

import <h4>

module barb13
          {
//*****************************************************************************
//          Global Declarations
//*****************************************************************************
          facility  joe;
          queue     joeq;

          rn_stream       Arrivals       seed=100000;
          rn_stream       Service        seed=200000;

          constant float  stop_time = 48000;

//*****************************************************************************
//          Customer Object
//*****************************************************************************
          object customer
               {
               actions
                    {
                    enqueue     joeq;
                    seize       joe;
                    depart      joeq;
                    advance     rv_uniform(Service, 12.0, 18.0);
                    release          joe;
                    terminate;
                    }
               }

//*****************************************************************************
//          Run Control
//*****************************************************************************
          procedure main
               {
               arrivals: customer
                    iat = rv_uniform(Arrivals, 12.0, 24.0)
                    until_time = stop_time;

               wait until (time >= stop_time and Q(joeq) == 0 and FNU(joe));

               report(system);
               exit(0);
               }

          }    // End of barb13 module
```