SLX Statistical Features

1.0 INTRODUCTION

This memo provides a description of experiment specification / statistics collection / analysis features which have recently been added to SLX. The remainder of this memo is organized as follows: In section 2, the architecture of SLX and our design goals for SLX, as they pertain to the features described herein, are reviewed. In section 3, a brief description is given of the way in which theses features have been implemented and packaged. In section 4, the major components are described. In section 5, individual declarations and statements are described. In section 6, individual utility procedures are described. Included are procedures for building and reporting confidence intervals, correlation, autocorrelation, etc. In section 7, built-in statistical functions, e.g., sample mean, are discussed. In section 8, random number streams are presented. In section 9, the random variate generators available in SLX are listed. Finally, in section 10, sample models are presented, with pertinent details described by line number.

2.0 SLX ARCHITECTURE AND DESIGN GOALS

SLX is a layered software system for discrete event simulation. Layer zero, the SLX kernel, is, roughly speaking, a dialect of C. SLX uses the syntax of C for expressions, assignment statements, looping constructs, if/else, etc. If you know C, you will be able to read and understand a great deal of the logic and computation of an SLX program. SLX adds to C a small number of very powerful, but easy-to-use primitives for discrete-event simulation. SLX omits features of C that are intended for systems programmers and features which, in our experience, have been sources of pitfalls. In sharp contrast to C, the run-time environment of SLX is totally secure (in the non-military sense). All computational errors, such as referencing beyond the end of an array or referencing invalid pointers, are trapped.

SLX includes extensibility mechanisms for introducing new statements into the language. SLX macros are a degenerate form of SLX defined statements. Both of these mechanisms are used extensively by SLX itself, but are also intended for use by end users. The metalanguage which is used to define an SLX defined statement or macro is SLX itself. This is very unusual. Most languages which include macro-like capabilities utilize a macro metalanguage which is different from the host language. For example, in C, #if, #ifdef, #else, and #endif preprocessor statements are used to achieve conditional macro expansion. The limited power of these statements pales in comparison to that of C itself. In SLX, virtually all of the non-simulation features can be used in a statement definition. For example, it’s even possible to read and write files during the expansion of a statement. Thus, SLX statement definitions can include arbitrarily complex logic.

3.0 IMPLEMENTATION OF STATISTICAL FEATURES

A handful of lowest-level statistical building blocks have been implemented as “hard-wired” SLX kernel features. For example, support for the GPSS/H-style “clear” statement (used for clearing statistics between experiments), was added at the kernel level. Some of the run-time library support, such as random variate routines, was written in C. The vast majority of the statistical code, however, was written in SLX. In order to provide a more user-friendly user interface, a number of SLX statements were defined as bridges between the user and lower-level procedural code. Most of these statements are expanded into procedure calls. While these calls could, in principle, be coded by hand, it's much easier to use the shorthand afforded by the defined statements. The defined statements were designed to look like the rest of SLX. Unless you look at a program listing in which statement expansions are displayed, you can't tell the difference between hard-wired, kernel-level statements, and those which have been added using the statement definition facility. The X in SLX stands for eXtensibility. The features described in this memo are themselves evidence of the efficacy of SLX as an extensible language. However, the extensibility mechanisms are designed to be used by the end user, so we hope that you will be able to add features to those we have provided, using our implementation as a paradigm.

As of this writing, a nice collection of statistical features has been implemented for SLX. Over the long haul, we'd like to do much more, but the time has come to get Release 1.0 out the door. We have tried very hard to avoid two kinds of errors in the design of these features. First, by exploiting the layered architecture of SLX, we have avoided implementing features as large, impenetrable black boxes. As presently constituted, the features described herein are contained in a file called stats.slx. In all probability, we will ship this source code as part of the final product, allowing you and/or your students (1) to see how we “did it,” and (2) to add code of your/their own. We do, however, reserve the right to hold back some of our source code, for obvious, proprietary reasons. The second kind of error we have tried to avoid is implementing features in such a way that precludes subsequent addition of features that we either did not foresee or that we foresaw, but didn’t have time to implement. For the most part, there's nothing special about the code we developed for statistical features; it's straightforward SLX code. Therefore, adding more of the same should be easy for both you and us.

4.0 MAJOR COMPONENTS

The most important components of SLX's statistical features are the random_variable, statistics, and interval object classes. A statistics object contains the data structures necessary to tabulate means, variances, minima, maxima, histograms, etc. A random_variable object contains information which describes basic characteristics of a random variable and includes an embedded statistics object. The embedded statistics object is used for tabulating the "master" statistics for the random variable (more on this later). A random variable can be unweighted, time-weighted, or user-weighted. Weights for time-weighted random variables, e.g., the length of a queue, are automatically calculated by SLX. User-supplied weights must be provided for each observation of a (non-time) weighted random variable. User-supplied zero weights elicit run-time warnings, and negative weights are treated as run-time errors.

Interval objects contain data structures which allow the collection of statistics over intervals of time. For example, one could define intervals for warmup and steady state statistics. The SLX "observe" statement is used to specify the interval(s) over which a random variable is to be observed. Intervals provide a second dimension to random variables. You can think of the statistics collected in an SLX model as a table whose rows are intervals and whose columns are random variables. The observe statement creates statistics objects and places them in the appropriate rows and columns of this "table." (In reality, the "rows" and "columns" are implemented as SLX sets.)

Intervals can be started and stopped. A warmup interval could be started at time zero and stopped at the time at which steady state were reached. Since the starting and stopping of intervals is performed with executable, rather than declarative code, the determination of what constitutes steady state could be implemented as a simple time delay or as arbitrarily complex logic. Stopped intervals can be restarted. For example, consider modeling a clinic in a hospital. Such facilities typically have resources, e.g., staff, which vary widely by shift, if not by hour. Thus, it might make sense to divide the day into 24 intervals and start and stop each interval repeatedly at the appropriate times of day for each day of a simulation.

The use of intervals does not impose as much overhead as you might think. Statistics which are collected over an interval (as opposed to "master" statistics which are always active) are calculated *relative to* the underlying master statistics. The only times interval-based statistics are updated, *per se*, are when (1) an interval is started or stopped and (2) when an interval-based statistic is evaluated. For example, evaluating expressions such as "mean(x over steady_state)" requires some on-the-fly updating.

5.0 DETAILED COMPONENT DESCRIPTIONS

5.1 Random variables

Random variables are declared by means of the random_variable statement. Note that this statement is defined in stats.slx as a language extension. It provides a convenient shorthand for defining a random_variable object. Examples of the random_variable statement follow:

```
random_variable          x, y, z;         // unweighted
random_variable(time)    qlength;         // time-weighted
random_variable(weight)  mystat;          // user-weighted

random_variable          x                histogram  start=0.0 width=10.0 count = 20;
random variable          daily_means      title = "Daily Means";
```

5.2 Intervals

Intervals are declared by means of the interval statement:

```
        interval                warmup          title = “Initial Transient Stats”,
                                hourly[0...23];
```

The observe statement is used to specify (random_variable, interval) relationships:

```
        observe                 x, y, z over warmup, steady_state;
```

Random variables are defined as an SLX *entity class*. (“Entity class” is GPSS/H parlance.) Without going into all of the ramifications of this implementation detail, let's explore one convenient consequence. Whenever an object is created for an object class that was defined as an SLX entity class, the created object is placed in a predefined set which contains all objects of the particular class. The name of the container set is the name of the entity class followed by “_set”. Thus, all random_variable objects are automatically placed into the a set named “random_variable_set”. If we want to observe *all* statistics over a collection of intervals, the following code can be used:

```
        pointer(random_variable)  rv;
        int                       i;

        for (rv = each random_variable in random_variable_set)
                for (i = 0; i <= 23; i++)
                        observe *rv over hourly[i];
```

This is a nice little example of the “nestability” of SLX features. Some day, we may have a GUI-based front end which can generate the kind of procedural code shown above. For the present, we must be content with having procedural functionality unmatched by other languages.

Intervals are started and stopped by means of the start_interval and stop_interval statements, respectively:

```
        start_interval  warmup;
        advance         1000.0;         // initial transient period elapses
        stop_interval   warmup;
        start_interval  steady_state;
```

Statistics collected over an interval are reset by means of the reset_interval statement:

```
        reset_interval  daily_stats;
```

The reset_interval statement zeros out all utilization integrals, counts, etc. for statistics based on the specified interval(s), and sets maxima and minima to the current values of the random variables. In practice, reset_interval is rarely used, because it discards information. Rather than collecting and discarding information, it’s easier to issue a start_interval statement at the point at which collection of statistics to be retained is reached. GPSS/H users should note that the reset_interval statement is very similar to GPSS/H's RESET control statement except (1) it affects only statistics collected over specified intervals, and (2) it does not affect “master” statistics.

5.3 Recording Observations of Random Variables

Observations are recorded by means of the tabulate statement (with homage to the GPSS/H TABULATE block):

```
        random_variable                 daily_mean;
        random_variable(time)           qtime;
        random_variable(weight)         mystat;
```

```
tabulate  daily_mean;               // unweighted
tabulate  qtime;                    // time-weighted
tabulate  mystat weight = w;        // user-weighted
```

6.0 UTILITY PROCEDURES

6.1 Introduction

Most utility procedures are provided in two forms, one which performs required computations and one which reports the result of such computations. For example, there's a build_mean_ci procedure for building a confidence interval for the mean of a random variable, based on independent replications, and there's a report_mean_ci procedure which reports the results. The calling sequence for the “report” procedure is simpler than that of the “build” procedure, so if all you want to do is *see* results, it's easier to use the “report” version. On the other hand, if you need to get your hands on the details, e.g., a resultant half-width, you can use the “build” form. Note that all arguments to “report” procedures are input arguments; i.e., report procedures do not modify their arguments.

6.2 Confidence Intervals Based on Independent Replications

The build_mean_ci procedure is used for building such confidence intervals. Its calling sequence is as follows:

```
procedure build_mean_ci(
        in  double      samples[*],     // observed samples
        in  int         scount,         // sample count
        in  double      level,          // confidence level, e.g., 0.95
        out double      smean,          // sample mean
        out double      stdev,          // sample standard deviation
        out double      half_width)     // resultant C.I. half-width
```

The report_mean_ci procedure is used for reporting such confidence intervals. Its calling sequence is as follows:

```
procedure report_mean_ci(
        string(*)       title,          // description
        double          level,          // confidence level, e.g., 0.95
        double          samples[*],     // sample data
        int             scount)         // sample count
```

6.3 Confidence Intervals Based on Antithetic Variates

Two vectors of (presumably) antithetic samples are averaged, a confidence interval is built, and the correlation between the two sample vectors is calculated. If the random number streams have been properly synchronized, and antithetic variates have been used, we should see some negative correlation in the outputs. The build_antithetic_mean_ci procedure is used for building such confidence intervals. Its calling sequence is as follows:

```
procedure build_antithetic_mean_ci(
        in  double      samples1[*],    // “thetic” samples
        in  double      samples2[*],    // antithetic samples
        in  int         scount,         // sample counts
        in  double      level,          // confidence level, e.g. 0.95
        out double      smean,          // grand mean of averaged samples
        out double      stdev,          // std dev of averaged samples
        out double      half_width,     // resultant half-width
        out double      corr)           // correlation between sample vectors
```

The report_antithetic_mean_ci procedure is used for reporting such confidence intervals. Its calling sequence is as follows:

```
procedure report_antithetic_mean_ci(
        string(*)       title,
        double          level,
        double          samples1[*],
        double          samples2[*],
        int             scount)
```

6.4 Confidence Intervals Based on Common Random Numbers (Differences of Means)

Sample vector two is subtracted from the sample vector one, a confidence interval is built, and the correlation between the sample vectors is calculated. If the random number streams have been properly synchronized, and common random numbers have been used, we should see substantial positive correlation. The build_common_mean_ci procedure is used for building such confidence intervals. Its calling sequence is as follows:

```
    procedure build_common_mean_ci(          // args as for antithetic
            in  double      samples1[*],
            in  double      samples2[*],
            in  int         scount,
            in  double      level,
            out double      smean,
            out double      stdev,
            out double      half_width,
            out double      corr)
```

The report_common_mean_ci procedure is used for reporting such confidence intervals. Its calling sequence is as follows:

```
    procedure report_common_mean_ci(
            string(*)       title,
            double          level,
            double          samples1[*],
            double          samples2[*],
            int             scount)
```

6.5 Confidence Intervals Based on Batch Means

Confidence intervals based on batch means are constructed by calling the build_batch_means_ci procedure. Its calling sequence is as follows:

```
    procedure build_batch_means_ci(in  double samples[*],
            in  int         scount,         // sample count
            in  int         batch_size,     // size of each batch
            in  double      level,          // confidence level, e.g. 0.95
            out double      smean,          // calculcated grand mean
            out double      stdev,          // standard devaiation of batch means
            out double      half_width)     // half width or resultant confifence interval
```

Confidence intervals for batch means are reported by calling the report_batch_means_ci procedure. This procedure requires specification of a smallest batch size, a largest batch size, and an increment between successive batch sizes. Results are summarized for each batch size tested. This allows quick comparison of a multiplicity of batch sizes. The calling sequence of report_batch_means_ci is as follows:

```
procedure report_batch_means_ci(
          string(*)       title,                  // descriptive title
          double          level,                  // confidence level, e.g., 0.95
          double          samples[*],             // samples to be batched
          int             scount,                 // sample count
          int             smallest_batch_size,
          int             largest_batch_size,
          int             batch_size_increment)
```

The autocorrelation observed between batches can be reported by calling the report_batch_means_auto-correlation procedure. While sample autocorrelations are typically poor estimators of true autocorrela-tions, this information provides a bit of insight into the independence, or lack thereof, of batches. The calling sequence for report_batch_means_autocorrelation is as follows:

```
procedure report_batch_means_autocorrelation(     // arguments same as above
          string(*)       title,
          double          level,
          double          samples[*],
          int             scount,
          int             sallest_batch_size,
          int             largest_batch_size,
          int             increment)
```

6.6 Correlation Between Sample Vectors

The sample correlation between two vectors of random samples is retuned by the correlation procedure. Its calling sequence is as follows:

```
procedure correlation(
          double   samples1[*],             // sample vector 1
          double   samples2[*],             // sample vector 2
          int      scount)                  // sample count

                   returning double
```

6.7 Autocorrelation of Samples

Sample autocorrelation at lags up to a specified maximum can be obtained by calling the build_autocorrelation procedure. Its calling sequence is as follows:

```
procedure build_autocorrelation(
          in  double      samples[*],             // vector of autocorrelated samples
          in  int         scount,                 // sample count
          in  int         max_lag,                // maximum lag to be evaluated
          out double      autocorrelation[*])     // autocorrelations at lags 1…max_lag
```

Sample autocorrelation can be reported by calling the report_autocorrelation procedure. Its calling sequence is as follows:

```
procedure report_autocorrelation(
          string(*)       title,                  // descriptive title
          double          samples[*],             // vector of autocorrelated samples
          int             scount,                 // sample count
          int             max_lag)                // maximum lage to be reported
```

7.0 BUILT-IN STATISTICAL FUNCTIONS

The built-in functions which are described below are functions of statistics objects. Recall that statistics objects represent a (random variable, interval) pair, e.g., “queue time over steady state.” While these functions can be called directly, macros which operate on (random variable, interval) pairs are also provided and offer better readability. The following built-in functions of random variables are provided:

Function	Meaning
rvs_sample_count	observation count
rvs_sample_min	sample minimum
rvs_sample_max	sample maximum
rvs_sample_sum	integral over time
rvs_sample_time_per_unit	integral divided by count
rvs_sample_mean	sample mean
rvs_sample_variance	sample variance

Note that sample_time_per_unit is biased, due to the fact that at the time it is evaluated, an operation reflected by the state of the underlying random variable may be incomplete. True time/unit statistics can be obtained only by tabulating elapsed times upon service completion. The bias of time/unit statistics in languages such as GPSS/H has been a dirty little secret for years. In SLX, with the ability to easily observe statistics over short intervals, e.g., hourly, the bias can become quite obvious.

The following macros are provided for (random variable, interval) pairs. In each case, if the “over interval” clause is omitted, “master” statistics (over time) are returned.

Macro	Meaning
sample_count(rv over interval)	observation count
sample_min(rv over interval)	sample minimum
sample_max(rv over interval)	sample maximum
sample_sum(rv over interval)	integral over time
sample_time_per_unit(rv over interval)	integral divided by count
sample_mean(rv over interval)	sample mean
sample_variance(rv over interval)	sample variance

8.0 RANDOM NUMBER STREAMS

Random number streams are implemented as SLX objects. For the present, we are using GPSS/H’s Lehmer generator, which has a period of 2^31 - 2. We are considering moving to a long period, composite generator. Which ever generator we choose will support the ability to calculate seed values quickly from requested offsets within the period of the generator. (We are carrying on GPSS/H’s policy of specifying seed *positions* rather than seed *values*.)

Random streams are declared by using the random_stream defined statement. Examples follow:

```
random_stream       arrivals,                             // default seed
                    service   seed = 100000,              // start position given
                    svc2      seed = 100000 antithetic;   // antithetic variates
```

Random seeds can be set by means of the rn_seed statement. (Remember that values are interpreted as offsets into the period of the generator). Examples follow:

```
rn_seed             arrivals  seed = 200000,              // user-specified
                    service   seed = new,                 // past the end of all other streams
                    svc2      seed = 100000 antithetic;   // antithetic variates
```

When “new” is requested, SLX calculates a starting position which is 1,000,000 samples byond the high water mark for all random streams used, rounded up to the next higher multiple of 1,000,000 for readability. Note that the combination “new antithetic” is not allowed. (Antithetic variates must be antithetic to previously used seed value.)

9.0 RANDOM VARIATE GENERATORS

SLX includes the following random variate generators. (We're still tinkering with calling sequences, so details are not given).

rv_beta
rv_binomial
rv_discrete_uniform
rv_erlang
rv_extreme_value_a
rv_extreme_value_b
rv_expo
rv_gamma
rv_geometric
rv_inverse_gaussian
rv_inverted_weibull
rv_bounded_johnson
rv_unbounded_johnson
rv_laplace
rv_logistic
rv_log_laplace
rv_log_normal
rv_negative_binomial
rv_normal
rv_poisson
rv_pearson5
rv_pearson6
rv_random_walk
rv_triangular
rv_uniform
rv_weibull

10.0 DISCUSSION OF SAMPLE PROGRAMS

10.1 NEWSTATX.SLX

This example illustrates the use of random variables, intervals, tabulation of random variables, and built-in macros for accessing collected statistics. It was used for desk checking the computed statistics. It is not based on a simulation model. In lines 5-7, three random variables are defined, an unweighted, a time-weighted, and a user-weighted. Each has a histogram. At line 12, a number of statistics collection intervals are defined. In lines 15-16, the observation of all statistics over the warmup, steady_state, and hourly[0] intervals is set up. In lines 18-19, two intervals are started. Note that, contrary to its name, hourly[0] is never stopped or restarted. (This is left as an exercise for the student.) In lines 21-24, a number of observations are tabulated. At line 28, a time-weighted random variable is tabulated, sandwiched between two time advances. In lines 32-33, the warmup collection interval is stopped, and the steady_state collection interval is started. In lines 35-42, additional observations are tabulated. At line 46, a report is issued to print all collected statistics. "system" is a set of container sets, with one container for each defined entity class. The report property for a set issues passes on the request to produce a report to each of the members of the set. Hence, "report system" reports all members of all defined entity classes. This is a close approximation to GPSS/H's standard output. In lines 52-113, individual statistics are printed.

10.2 BARBANTI.SLX

This program makes 100 independent runs of a simple barbershop, followed by 100 runs using antithetic variates. In lines 16-17, random number streams are defined for interarrival and service times. At line 19, the single random variable of interest, qtime, is defined. The outer loop through the first 100 replications begins at line 37. In lines 39-43, seed values for each replication are established and incremented for the next replication. In lines 53 and 57, the status is printed for the random number streams after the first and last replications, respectively. In lines 59-60, local variables are assigned the original random number stream positions. The outer loop through 100 antithetic replications begins at line 62. In lines 64-68, antithetic seeds are established for each replication. As above, the status of the random number streams is reported after the first and last antithetic replications. In lines 84 and 85, confidence intervals based on independent replications are printed for the 100 "thetic" replications and the 100 antithetic replications. In lines 87-88, a confidence interval is reported for the averaged thetic and antithetic means.

10.3 BARBCOMN.SLX

BARBCOMN.SLX illustrates the use of common random numbers. It compares two configurations of the barbershop. In the second configuration, mean service times are longer by one minute. The structure of this program is very similar to BARBANTI.SLX, except that common random numbers are used, and a confidence interval is built for the difference (subtracting the second sample vector from the first) between samples. Just for the fun of it, a confidence interval is also built for two sets of replications for which common numbers were *not* used. Note the warning message about the lack of correlation in the non-common case.

10.4 BARBBM.SLX

BARBBM.SLX illustrates the use of batch means. The model used is a simple barbershop, and the statistic of interest is time-in-queue. The model is terminated at a predetermined time, resulting in 6,664 correlated observations of time-in-queue. In lines 39-50, a variety of very interesting reports is produced. Batch means for batches of sizes 5…200, in increments of 5 are displayed. The autocorrela-tions for lags 1-20 are shown. We all know that sample autocorrelations tend to be poor estimators of true autocorrelations, but in this case, the number of samples is large enough to get good estimates. In any case, for students witnessing this behavior for the first time, the message is pretty clear, regardless of the precision of the results. Finally, autocorrelations between successive batch means are shown for 1ags 1-20 for three different batch sizes.

10.5 TJSLOCK2.SLX

TJSLOCK2.SLX provides the capstone of this memo. This problem was posed by Tom Schriber in *Simulation News Europe* as a modeling challenge. For those of you who have not seen the problem statement, I can provide one. The problem explores the operation of a canal-and-lock system. What makes the system interesting is that the lock can accommodate only one ship at a time, and the canals entering and exiting the lock are wide enough for only one barge at a time. The model explores alternative strategies for grouping eastbound and westbound barges into batches, to improve efficiency. The problem serves as an ideal vehicle for illustrating the use of antithetic variates and common random numbers. Tom's problem statement requires making antithetic replications and building confidence intervals for an initial configuration of the system, and it requires the use of common numbers and building difference-of-means confidence intervals comparing the original configuration with a proposed configuration.

Neither the model, nor the procedure run_model, which supervises the running of a specified number of replications of the model, will be discussed in this memo. Our discussion will focus on the design of experiments and statistical output requested in the main program (lines 165-259). In lines 169-170, vectors are defined for storing sample means for a variety of replications. In lines 174-190, three independent sets of 100 replications are run for the initial configuration of the system. For each set of replications, a confidence interval is reported. In lines 189-211, three sets of 50 replications, antithetic to the first 50 replications of the three sets described above, are run. For each paired set of antithetic replications, a confidence interval is reported. In lines 215-218, 50 replications of a proposed configuration are run, and a confidence interval is built for the difference between the means of the new and old configurations. Because the random numbers are intentionally unsynchronized, there is a nearly total lack of correlation between the paired means. Note the warning issued in the output. In lines 220-232, this pattern is repeated twice more, with similar results. In lines 234-253, we finally get around to doing it right, i.e., building confidence intervals for the difference-of-means, using common random numbers. The results are quite dramatic. Finally, in lines 255-258, a histogram is built and reported to explore "just how normal" the distribution of the sample mean appears. The results of this little exercise bring us back to reality.

11.0 CONCLUSIONS

The features, examples, and results reported in this memo reflect the current status of SLX. The statistical capabilities which have been implemented to date are those we felt were most important to practitioners and students of simulation. The lack of a point-and-click interface is counterbalanced by the tremendous ease with which one can get one's hands on things. The "impenetrable black box" syndrome has been avoided. To date, SLX's statistical features have had little use outside Wolverine; however, based on our prior experience with SLX testers, we can offer some predictions. What we've found through interaction with our SLX testers is that they have done two things to a greater extent than we anticipated. First, they have made extensive use of lowest-level primitives. Our prior experience with GPSS/H lead us to expect that users would be more comfortable operating at higher language levels. Second, nearly everyone has made use of the statement/macro definition capabilities to custom tailor SLX to meet their specific needs. Of course, these two tendencies are anything but independent. Since the statistical capabilities of SLX have been implemented in the same spirit as the rest of SLX, we anticipate that users will do the same sorts of things with statistical constructs that they have done with SLX's modeling constructs. We look forward to seeing the results.