Getting Started:
CSIM19 Simulation Engine
(C++ Version)

Mesquite Software, Inc.
8500 N. Mopac, #825
Austin, TX 78759
(512) 338-9153
Internet: info@mesquite.com

To navigate through this document, you may either select from the list of bookmarks to the left, or you may click on sections in the Table of Contents page.

Table of Contents

Introduction to CSIM19 for C++ Programmers

Introduction

CSIM19 is a library of routines, for use with C or C++ programs, which allows you to create process-oriented, discrete-event simulation models. This guide leads you through a simple model which uses the CSIM19 routines. It closes with a brief discussion of the CSIM[+] objects used to implement more complex models.

Example

The most basic simulation model is a single server and queue with arriving customers. With certain restrictions, this is the well-known M/M/1 queue. In the CSIM19 version of this model, there is a facility consisting of a single server and a single queue. In addition, there is a source of customers. As a customer arrives, it either seizes (uses) the server if it is free (not in use) or it joins a queue of waiting customers if the server is already busy (in use). When one

[+] Copyright by Microelectronics and Computer Technology Corporation, 1987 - 1994 customer leaves the server, the next customer in the queue begins to use the server.

The key parameters in such a model are:

- The intervals of time between customer arrivals
- The intervals of server usage

The results of a study of such a model are:

- The average customer response time (time of arrival to time of departure)
- The customer throughput rate (customers served per unit time)
- The server utilization (percentage of elapsed time that the server is busy)
- The average queue length (number of customers at the facility)

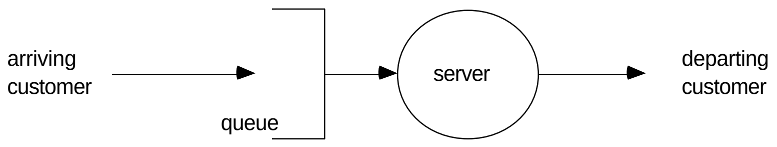

Figure 1: A Single Server Queue

A CSIM19 program (in C) to model this simple system is as follows:

```
/*this CSIM program simulates an M/M/1 service center*/

#include <cpp.h>                /*include the CSIM C++ header file*/

facility *f;                    /*the service center*/

extern “C” void sim()           /*sim process*/
{
    create("sim");              /*make this a process*/
    f = new facility("f");      /*create the service center - f*/
    while(simtime() < 5000.0) { /*loop until end of simulation*/
        hold(exponential(1.0)); /*delay between customer arrivals*/
        customer();             /*generate new customer*/
        }
    report();                   /*produce statistics report*/
}

void customer()
{
    create("customer");         /*make this a process*/
    f->use(exponential(0.5));   /*obtain needed amount of service*/
}
```

The CSIM output for this example is as follows:

```
CSIM Simulation Report (Version 19 for MSVC++)

                Mon May 13 13:42:39 1996


        Ending simulation time:          10001.909
        Elapsed simulation time:         10001.909
        CPU time used (seconds):         0.490

FACILITY SUMMARY

facility  service  service         through-  queue    response  compl
name      disc      time    util.    put    length      time    count
-----------------------------------------------------------------------
f         fcfs    1.00954   0.512  0.50680  1.01983   2.01229    5069
```

This example shows most of the important features of a CSIM model:

1. In this example, there are two processes:

   a. The base "sim" process which initializes the model and generates the customer arrivals at varying interarrival intervals, and

   b. The "customer" process, which mimics the behavior of a customer of the f facility. Notice that there can be several customers (instances of the customer process) "active" at the same time: one using the server and others arriving and waiting in the queue.

2. A CSIM process is a C++ procedure which executes the "create" statement. Executing a create statement does two things:

   a. Establishes the procedure which executes the statement as an independent, ready-to-run process, and

b. Returns control to the calling process.

3. A facility is declared with the "FACILITY" statement and is initialized by the "facility()" function.

4. The CSIM variable "clock" contains the current simulated time (the value of the simulated clock). In CSIM, time is a double precision, floating point value.

5. The "hold" statement causes time to pass for the process executing the statement; in the example, the "hold(exponential (1.0));" statement models the intervals of time between customer arrivals.

6. In many simulation models, it is appropriate to specify sequences of time intervals with probability distributions. In the M/M/1 queue, the interarrival intervals and the service intervals are "sampled" from negative exponential distributions. In CSIM, the exponential() function gives such samples.

7. Use of the facility is modeled by the "f->use(exponential (0.0)) statement (in this case, the facility being used is f).

8. To elaborate on an earlier point, there may be multiple instances of the customer process active and competing for use of the server at the same time. Modeling parallel activities such as this is a major feature of process-oriented models such as those implemented with CSIM.

The interactions among the processes in a model can be seen by looking at an activity (debug) trace which is generated by CSIM during the execution of a model (a trace is generated only upon request). A segment of the activity trace for the sample model is shown below:

```
 time       process     .id  priority  status
---------------------------------------------------------------
0.000         sim         1      1      create sim 1
0.000         sim         1      1      join class default
0.000         sim         1      1      facility f with 1 server
0.000         sim         1      1      sched proc: t = 0.000, id = 2
0.000         sim         1      1      create customer 2
0.000       customer      2      1      join class default
0.000         sim         1      1      hold 1.332
0.000         sim         1      1      sched proc: t = 1.332, id = 1
0.000       customer      2      1      use facility f, t = 1.739
0.000       customer      2      1      reserve f
0.000       customer      2      1      hold 1.739
0.000       customer      2      1      sched proc: t = 1.739, id = 2
1.332         sim         1      1      sched proc: t = 0.000, id = 3
1.332         sim         1      1      create customer 3
1.332       customer      3      1      join class default
1.332         sim         1      1      hold 2.351
1.332         sim         1      1      sched proc: t = 2.351, id = 1
1.332       customer      3      1      use facility f, t = 0.626
1.332       customer      3      1      dequeue facility f
1.739       customer      2      1      terminate
1.739       customer      3      1      hold 0.626
1.739       customer      3      1      sched proc: t = 0.626, id = 3
2.365       customer      3      1      release f
2.365       customer      3      1      terminate
 .
```

In this activity trace segment, we can see the following sequence of simulated activities:

- The base (first) process, sim, starts at time 0.000 by initializing the facility (named f), starts the first customer process and then does a hold of 1.332 time units (the interval until the next customer arrival which is generated by using a negative exponential distribution with a mean of 2.0). Since this example doesn’t divide its processes into different classes for reporting purposes, all processes are shown as joining the default class. This example also does not assign explicit priorities to processes, so they all default to priority 1.
- Because sim has "suspended" execution, the first customer process can begin execution, also at time 0.000 (the id of this instance of customer is 2, so we will refer to it as customer.2). Upon arrival, customer.2 executes a "use", which reserves the facility f. Since f is free, customer.2 gets it and then does a hold, simulating its service interval of 1.739 time units (generated by using a negative exponential distribution with a mean of 1.0).
- At time 1.332, the hold for sim expires, so sim resumes execution and generates the arrival of the next customer, customer.3. sim then does a hold of 2.351 units of time, the interval until the next customer arrival.
- Customer.3 begins execution at 1.332. It tries to use f, but f is busy (it is being held by customer.2), so customer.3 must wait until customer.2 completes its service interval.
- At time 1.739, customer.2 finishes its service interval, so it releases f. This frees f for use by the next customer in the queue of waiting customers. Since customer.2 is finished, it terminates. Termination for a process is automatic when the process (procedure) does a normal procedure exit.
- Customer.3 is able to proceed (its reserve has succeeded), so does a hold for its service interval (0.626 time units).

- At time 2.365, customer.3 completes its service interval, releases f and terminates.
- Sim is still holding, simulating the interval until the arrival of the next customer.
- The model will continue this sequence of activities (customer arrivals, requesting the facility, etc.) until the value of clock exceeds 10000.0 (the length of the simulated experiment specified by the define constant SIM_TIME). When the experiment finishes, the CSIM report is printed (by the report() procedure) and sim exits, causing the program to end.

The CSIM report in this example gives a statistical summary of the usage of the f facility by the 5069 customer processes which completed service during the 10001.909 simulated time units covered by the experiment. In the report, we can see the following:

- The mean service interval at the facility is 1.010 time units. The fact that it is not 1.0 results from the use of the samples from the negative exponential probability distribution with mean 1.0.
- The utilization of the facility is 0.512. This is the percentage of the elapsed time during which the server at f was busy (in use).
- The throughput rate is 0.5 customers per unit time.
- The mean queue length is 1.020. This is the average number of customers at the facility, including both customers at the server and in the queue of waiting customers.
- The average response time experienced by customers at the facility (resp) is 2.012 time units. The response time includes both time in the queue and time at the server.
- The number of completed customers at the facility is 5069.

CSIM Objects

CSIM provides a complete set of objects which can be used to construct models of almost any kind of system, at any level of complexity and detail. The objects supported by CSIM are:

- Process - used to model elements of the workload, clients and servers, or any other active components of the system
- Facility - used to model resources which are seized (used) by processes
- Storage - used to model resources which are are partially allocated to processes
- Buffer– uses to model buffers with finite capacity
- Event - used to synchronize and control interactions between processes
- Mailbox - used to exchange information between processes
- Tables, Qtables, Meters, and Boxes - used to collect explicit statistics (note: statistics on usage of facilities and storage blocks are collected automatically)
- Process class - used to segregate facility usage statistics
- Stream of random numbers - used to generate multiple streams of samples from specified probability distributions

These objects can be created and used by the program to give accurate and detailed insights into the structure, organization and behavior of complex systems.

For more information on how to do this and how to derive many benefits from building and using CSIM models, contact Mesquite Software, Inc.

CSIM19 Tutorial (C++ Version)

Introduction

This section gives an overview of all of the important objects and other features of the CSIM19 library. A CSIM[+] model is a C++ (or C) program that uses the functions and procedures in the CSIM19 library to implement process-oriented, discrete-event simulation models. Each model will mimic the behavior of the system being modeled. Using a model helps the user analyze the behavior of a system and can lead to improvements in the operation and performance of that system.

It is assumed that the reader of this document has a working knowledge of the C++ programming language and is familiar with the concepts of discrete-event simulation models.

[+] Copyright by Microelectronics and Computer Technology Corporation, 1987 - 1994

Processes

A CSIM process is used to model the active elements of a system. These could include elements of the workload, clients and servers found in the system, and other components that are active parts of the system model. In CSIM, a process is a C++ procedure that executes the "create()" statement. Every time a "create" statement is executed, a new instance of that process is created. Each instance of a CSIM process has the following attributes:

- Its own internal state (local variables and registers)
- A unique process id
- A process priority
- One of the following external states:
  - Executing
  - Waiting-to-execute
  - Holding (while some period of time elapses)
  - Waiting (for some event to occur)

CSIM processes should not be confused with processes in the platform operating system (such as UNIX) or operating system supported threads (such as lightweight threads in SunOS). The concepts are similar, but the implementations are separate.

It is very <u>important</u> to understand the *flow of control* that is used by CSIM processes. When a newly called procedure executes a create() statement, the following actions occur:

1. A process control block (pcb) for the new procedure (really the new process) is created and put on the “next event list”, and

2. Control is returned immediately to the process which invoked this new process.

So, after a new process is “called”, the old process is *still* executing and the new process will execute only after the current (old) process “gives up” (e,g., does a *hold* or a statement which results in a wait).

For example, after this CSIM code fragment has executed:

```
...
for(i = 0; I < 100; I++)
        customer(i);
hold(1000.0);
...
```

100 instances of the process named “customer” will be created, but *none* of them will start to execute until the calling process executes the *hold* statement.

Facilities

Facilities are those objects which processes “use” or occupy. They can be defined as:

- A single server facility (can only service one process at a time)
- A multi-server facility (can service n processes at once, where n is the number of servers defined for the facility)
- An array of single server facilities

Each facility is given a name, which is used solely for output (reports, status, and traces).

By default, a facility services processes in priority order. Where multiple processes have the same priority, they will be served on a first-come-first-served basis. A number of other service disciplines can be specified.

The following examples show how a facility can be used from within a process.

- *To declare, initialize, and use a facility with a single queue and a single server:*

```
facility *single_server;                /* declare facility variable/
...
single_server = new facility("sngle srvr");   /* initialize facility named sngle srvr*/
...
single_server->use(service_time);       /* use facility for length of service_time*/
...
single_server->reserve();               /* reserve (use) facility */
...
hold(service_time);
...
single_server->release();               /* release the facility  */
...
```

The "use()" statement and the sequence "reserve(), hold(), release()" behave in similar ways; the only difference is when (in simulated time) the service_time variable is evaluated. By convention, the "use()" statement is used when the process will be "using" the facility, while the "reserve" statement is used when the process will acquire exclusive use of the facility and then do something other than a "hold" statement.

Processes are ranked in the queue of waiting processes in order of their process priorities, with the highest priority at the head of the list. In the case of equal priorities, the process doing the earliest reserve is ahead of processes doing reserves at later points in time. If all reserving processes have the same priorities, then the resulting scheduling policy (discipline) is first-come, first-served (or FIFO - first in, first out).

- *To declare, initialize, and use a facility with a single queue and three servers:*

```
const long NUM_SRVRS = 3;                /* set number of servers to 3 */
facility_ms *multi_server;               /* declare facility_ms object ptr */
...
multi_server = new facility_ms("multi srvr", NUM_SRVRS);/*initialize srvr with 3 srvrs*/
...
multi_server->use(service_time);         /*use facility for length of service_time*/
….
```

- *To declare, initialize, and use an array of ten single server facilities:*

```
const long NUM_FACS = 10;               /*set number of facilities in array to 10 */
facility_set *facs;                     /* declare facility array */
...
facs = new facility_set("facs", NUM_FACS);   /*initialize set of 10 facilities */
i = random(0, NUM_FACS-1);              /*select the facility to be used next*/
(*facs)[i].use(service_time);           /*use facility[i] for length of service_time*/
```

- *To reserve a facility only if it can be obtained within a given length of time:*

```
const double  TIME_OUT = 5.0;        /*set length of time to wait for facility*/
.....
st = single_server->timed_reserve(TIME_OUT);   /*reserve facility in 5 time units*/
if(st != TIMED_OUT) {                /*if facility was, in fact, reserved in time*/
     hold(service_time);             /*simulate servicing customer for service_time*/
     single_server->release();       /*release facility since service is now complete*/
} else {                             /*request timed out */
....
     }
```

- *To declare, initialize, and use a synchronous facility (a synchronous facility is one in which reserves are granted only at regular points in time (called clock ticks)):*

```
const double PHASE = 0.5;        /*set time to onset of first clock cycle to 0.5 */
const double PERIOD = 1.0;       /*set length of clock cycle to 1 time unit*/
faciolity *bus;                  /*declare facility variable bus  */
...
bus = new facility("bus");       /*initialize facility and name it bus */
bus->synchronous(PHASE, PERIOD); /*make the facility synchronous */
...
bus->reserve();                  /*reserve the facility  */
...
bus->release();                  /*release facility since process no longer needs it*/
...
```

- *To define the preempt-resume service discipline for a facility:*

```
FACILITY cpu;                    /*declare facility variable cpu */
...
cpu = fnew acility("cpu");       /*initialize facility and name it cpu */
cpu->set_servicefunc(pre_res)    /*set service protocol to preempt-resume*/
...
priority = 100;                  /*make process high priority */
cpu->use(service_time);          /*preempt lower priority process and use facility*/
...
```

It is important to notice that when scheduling disciplines other than first-come, first-served are in use at a facility, then the "use()" method is the only way to make use of the facility. This means that a process cannot reserve such a facility and then do something other than use that facility.

Storages

A CSIM storage is a resource which can be partially allocated to a requesting process. A storage consists of a counter (to indicate the amount of available storage) and a queue for processes waiting to receive their requested allocation. A storage set is an array of these basic storages.

A storage can be designated to be synchronous. In a synchronous storage, each allocate is delayed until the onset of the next clock cycle.

Each storage must be given a name, which is used solely for output (reports, status and traces).

The following examples show how storage can be used from within a process.

- *To declare, initialize, and use a storage:*

```
const long STORE_AMT = 100;          /*set amount of storage to 100 units */
storage *mem;                        /*declare storage variable mem */
...
mem = new storage("mem", STORE_AMT);/*initialize storage named mem with 100 units */
...
amt = random(1, STORE_AMT);          /*decide how much storage to allocate this time */
mem->alloc(amt);                     /*get amount of storage decided upon*/
...
mem->dealloc(amt);                   /* release storage which is no longer needed */
...
```

- *To declare, initialize, and use an array of five storage blocks:*

```
const long NUM_STORES = 5;            /*set number of storage blocks in array*/
const long STORE_AMT = 100;           /*set amount of storage in each storage block*/
storage _set *mems;                   /*declare storage block array*/
...
mems = new storage_set("mem", STORE_AMT, NUM_STORES);  /*initialize stor mem with 100 units
per block*/
...
amt = random(1, STORE_AMT);           /*decide how much storage to allocate */
(*mems)[3].alloc(amt);                /*get storage from the fourth storage block*/
...
(*mems)[3].dealloc(amt);              /*release storage which is no longer needed*/
....
```

- *To get storage only if it can be obtained within a given length of time:*

```
...
st = mem->timed_alloc(amt, 1.0);      /*get storage if possible within 1 time unit */
if(st != TIMED_OUT) {                 /*if storage was gotten within the time limit */
...
    mem->dealloc(amt);                /*release storage which is no longer needed */
}
else {                                /* allocate timed out  */
...
}
```

- *To declare, initialize, and use storage synchronously (allocations will take place only at regular points in time (called clock ticks)).*

```
const double PHASE = 0.5;             /*set time to onset of first clock cycle to 0.5*/
const double PERIOD = 1.0;            /*set length of clock cycle to 1 time unit*/
const long STORE_AMT = 100;           /*set amount of storage in block to 100 units*/
storage *mem;                         /*declare storage variable mem */
mem = new storage("mem", STORE_AMT);/*initialize storage named mem with 100 units*/
mem->synchronous(PHASE, PERIOD);      /*make storage allocations synchronous*/
...
mem->alloc(5);                        /*get 5 units of storage for this process */
...
mem->dealloc(5);                      /*release storage which is no longer needed*/
…
```

Buffers

A CSIM buffer is a resource which can be partially allocated to a requesting process. A buffer consists of a counter (to indicate the number of slots, represented as tokens, in the buffer) and a two queues, one for processes waiting to get tokens from the buffer, and one for processes waiting for space to put (or return) tokens to the buffer.

Each buffer must be given a name, which is solely used for output (reports, status and traces).

The following examples show how a buffer can be used from within a process

- *To declare, initialize and use a buffer*

```
const long BUFFER_AMT = 100;
buffer *buf;

buff = new buffer("buff", BUFFER_AMT);

amt – random(1, BUFFER_AMT);
buff->get(amt);
…
buff->put(amt);
```

- *To get space in a buffer only if it can be obtained with a given length of time:*

```
st = buff->timed_get(amt, 1.0);
if(st != TIMED_OUT) {
        …
        buff->put(amt);}
else { … }
```

Events

Events are used to synchronize and control interactions between different processes. A CSIM event has two states: occurred (OCC) and not occurred (NOT_OCC). A process can either *wait* for an event to occur or *queue* on the event.

If a process "waits" for an event:

- If the event is in the *not occurred* state, the process is suspended and placed in a queue of processes waiting for the event to happen (occur). When some other process does a "set" operation on that event, all of the waiting processes are re-activated (allowed to proceed) and the event is reset to *not occurred.*
- If the event is in the occurred state, the process continues to execute and the event state is changed to *not occurred*.

If a process queues on an event:

- If the event is in the *not occurred* state, the process, is suspended and placed in a queue of processes queued for the event to happen (occur). When some other process does a "set" operation on that event, only the first queued process is re-activated (allowed to proceed) and the event is reset to *not occurred.*
- If the event is in the occurred state, and there are no other processes queued on that event, the process continues to execute and the event state is changed to *not occurred*.

Events can be defined as either an individual event or an array of events.

Each event must be given a name, which is used solely for output (status and traces).

The following examples show how events can be used from within a process.

- *To declare, initialize, and use an event:*

```
event *ev;             /*declare event variable ev */
...
ev = new event("ev");   /*initialize an event named ev */
....
ev->wait();            /*wait for event to occur before proceeding */
...
ev->queue(); /*wait for event to occur, for processes to respond before proceeding*/
ev->set();             /*indicate that an event has occurred */
...
```

- *To monitor an event, to collect statistics on its use*

```
ev->monitor();   /* invoke statics collection for ev */
```

- *To declare, initialize, and use an array of twenty-five events:*

```
const long NUM_EVENTS = 25;                    /*set number of events in array */
event_set *ev_arr;                             /*declare event array */
....
ev_arr = new event_set("ev arr", NUM_EVENTS);    /*initialize array of 25 events*/
....
(*ev_arr)[5].wait();        /*wait for sixth event to occur before proceeding */
...
(*ev_arr)[5].set();         /*indicate that sixth event has occurred*/
...
```

- *To wait or queue for any event in an array of events to happen:*

```
i = ev_arr -.wait_any();       /*i is index of event which occurred */
     OR
i = ev_arr->queue_any();       /*i is index of event which occurred */
...
....
```

- *To wait for an event only if it occurs within a given length of time:*

```
…
st = ev->timed_wait(50.0);          /*wait for a maximum of 50 time units */
if(st ! = TIMED_OUT) {              /*did not timed out */
……
}
```

Mailboxes

A mailbox allows for the synchronous exchange of data between CSIM processes. Any process may send a message to any mailbox, and any process may attempt to receive a message from any mailbox.

A mailbox is comprised of two FIFO queues: a queue of unreceived messages and a queue of waiting processes. At least one of the queues will be empty at any time. When a process sends a message, the message is given to a waiting process (if one exists) or it is placed in the message queue. When a process attempts to receive a message, it is either given a message from the message queue (if one exists) or it is added to the queue of waiting processes.

A message can be either a single integer or a pointer to some other data object. If a process sends a pointer, it is the responsibility of that process to maintain the integrity of the referenced data until it is received and processed.

Each mailbox must be given a name, which is used solely for output (status and traces).

The following examples show how mailboxes can be used from within a process.

- *To declare, initialize, and use a mailbox:*

```
mailbox *mb;            /*declare mailbox variable mb */

long msg_r,msg_s;       /*message variables */
```

```
...
mb = new mailbox("mb");      /*initialize a mailbox named mb */
...
mb->receive(&&msg_r);        /*receive message (in msg_r) from mailbox mb */
...
mb->send(mb, msg_s);         /*send message in msg_s to mailbox mb */
...
```

A message is a single variable. It can be either an integer or a pointer to a (message) structure.

- *To monitor a mailbox, to collect statistics on its use*

```
mb->monitor();
```

- *To wait for a message only if it comes in within a given length of time:*

```
...
st = mb->timed_receive(&msg_r, 100.0);    /*wait for a maximum of 100 time units */
if(st ! = TIMED_OUT) {                    /*if not timed out */
....
}
```

- *To declare, initialize and use an array of twenty-five mailboxes:*

```
const long NUM_MBOXES = 25;
mailbox_set *mbox_arr;
. . . .
mbox_arr = new mailbox_set(“mbox set”, NUM_MBOXES);
```

- *To receive a message from any mailbox in an array of mailboxes:*

```
i = mbox_arr->receive_any(&msg);
```

- *To send a message to a mailbox which is member of an array of mailboxes*

```
(*mbox_arr)[3].send(msg);
```

- *To receive a message from any mailbox in an array of mailboxes within a specified interval of time:*

```
st = mbox_arr->timed_receive_any(&msg, 1.0);
if(st != TIMED_OUT) {
                         // process message
} else {
                         // deal with time out
}
```

- *To send a message and wait until the message is received*

```
mb->synchronous_send(msg);
```

- *To send a message and wait until the message is received within a specified interval of time*

```
st = mb->timed_synchornous_send(msg, 1.0);
if(st == != TIMED_OUT) {
                                 // message received OK
} else {
                                 // message not received
}
```

Tables and Qtables

CSIM automatically collects some usage statistics. In order to allow the user to collect other statistics describing different aspects of the behavior of the system, CSIM supplies several objects:

- Table - collects floating point values and then gives a statistical summary consisting of, as shown:

```
TABLE 1:  table

     minimum        0.000016        mean                 1.000040
     maximum       10.336942        variance             0.999862
     range         10.336926        standard deviation   0.999931
     observations      10000        coefficient of var   0.999890
```

- A histogram can be specified for a table in order to obtain more detailed information about the recorded values. A histogram has a user-defined number of intervals and minimum and maximum values. The histogram will actually create two more intervals than specified, one for values less than the minimum and one for values greater than or equal to the maximum. A histogram report consists of a line of output for each interval, as shown:

```
                                                cumulative
   lower limit      frequency    proportion    proportion

       0.00000           6322      0.632200      0.632200   ********************
       1.00000           2288      0.228800      0.861000   *******
       2.00000            878      0.087800      0.948800   ***
       3.00000            322      0.032200      0.981000   *
       4.00000            112      0.011200      0.992200   .
       5.00000             48      0.004800      0.997000   .
       6.00000             22      0.002200      0.999200   .
       7.00000              5      0.000500      0.999700   .
       8.00000              1      0.000100      0.999800   .
       9.00000              1      0.000100      0.999900   .
  >=  10.00000              1      0.000100      1.000000   .
```

- Confidence intervals can also be specified for a table (see page 36).
- Qtable - tracks state changes (for example the number of processes in a queue). The qtable reports on the following items:

```
QTABLE 1:  qtable

      initial       0     minimum       0     mean                  0.795029
      final         0     maximum       7     variance              0.802270
      entries   10000     range         7     standard deviation    0.895696
      exits     10000                         coeff of variation    1.126620
```

- A histogram can be specified for a qtable. It gives more detail on the time spent in each state.
- Confidence intervals can be specified for a qtable (see page 36).

```
                                         cumulative
   number    total time   proportion   proportion

        0    3523.17812    0.352291     0.352291   **************
        1    5078.27616    0.507790     0.860081   ********************
        2    1306.88320    0.130679     0.990759   *****
        3      90.75115    0.009074     0.999834   .
        4       1.66151    0.000166     1.000000   .
```

Tables can be defined to be either permanent or non-permanent. A permanent table is not affected by requests to reset statistics or rerun the model, and can thus be used to gather data across multiple runs of a model.

Each table must be given a name, which is used solely for output (reports, status, and traces). The tables can be printed using various report statements.

The following examples show how tables and qtables can be used:

- *To declare, initialize, and use a non-permanent table with a histogram:*

```
table *tbl;                         /*declare table variable tbl */
...
tbl = new table("tbl");             /*initialize a table named tbl */
tbl->add_histogram(10,0.0,20.0);    /*add a historgram to a table named tbl */
...
t = clock;                          /*get current time */
single_server->reserve();           /*reserve a single server facility */
     x = clock - t;                 /*calculate time spent on queue (delay  interval)*/
     tbl->tabulate(x);              /*record delay interval in table  */
...
```

- *To declare, initialize, and use a non-permanent qtable with a histogram::*

```
qtable *qtbl;                   /*declare queue histogram and table variable  hst*/
...
qtbl=qtable("qtbl");
qtbl->add_histogram(20,0,20);

qtbl->note_entry();             /*record entry onto queue for facility */
single_server->reserve();       /*reserve a single server facility */
     qtbl->note_exit();         /* record exit from queue for facility  */
     hold(exponential(2.5));
```

- *To add confidence intervals to a table and to a qtable:*

```
table_confidence(tbl);              /* add confidence interval */
qtable_confidence(qtbl);            /* add confidence interval */
```

Meters and Boxes

Meters are used to gather statistics on the rate at which entities flow past a point as well as the times between passages. Meters can be used to measure arrival rates, completion rates, allocation rates, and interpassage times.

- A meter report gives the following information:

```
METER 1:  meter

     count            10000        rate               0.989904

     interpassage time statistics

     minimum       0.000144        mean               0.999140
     maximum       9.135145        variance           1.010617
     range         9.135002        standard deviation 1.005294
     observations     10000        coefficient of var 1.006159
```

- Histograms can be added to meters.
- Confidence intervals can also be used with meters.

A box conceptually encloses part or all of a model. This box gathers statistics on the number of entities in the box and on how much time they spend in the part of the model deliniated by the box. Boxes are used to gather statistics on queue lengths, response times, and populations.

- A box report gives the following information:

  Statistics on elapsed times (see tables):

  - minimum, etc.

  Statistics on population variation (see qtables):

  - initial, etc.

- Histograms can be added to boxes, (for both elapsed times and population).

- Confidence intervals can also be used with boxes, (for both elapsed time and population).

```
BOX 1:  box

     statistics on elapsed times

     minimum        0.000037       mean                 0.784577
     maximum        6.498131       variance             0.622643
     range          6.498094       standard deviation   0.789077
     observations      10000       coefficient of var   1.005736

     statistics on population

     initial      0     minimum     0     mean                 0.776656
     final        0     maximum     6     variance             0.775737
     entries  10000     range       6     standard deviation   0.880759
     exits    10000                       coeff of variation   1.134040
```

- *The following example shows how to declare, initialize, and use a meter:*

```
meter *mtr;                       /* declare meter variable m */

mtr = new meter("mtr");           /* initialize a meter named mtr */
```

```
mtr->note_passage();            /* note passage of process */
```

- *The following example shows how to declare, initialize, and then use a box*

```
box *b;                         /* declare box variable b */

b = new box("b")                /* initialize a box named b */

timestamp = b->enter();         /* note enter box */

b->exit(timestamp);             /* note exit */
```

Confidence Intervals

A confidence interval for a statistic is a range of values in which the true “answer” is believed to lie with a high probability. In a simulation model, an output, e.g.. system response, can be an important statistic and we would like to calculate a confidence interval for this statistic so that we can assess it’s statistical accuracy. In CSIM19 we can add confidence interval calculations to tables, qtables, meters, and boxes.

- The report for a data collection object with confidence intervals is as shown:

```
confidence intervals for the mean after 10000 observations

     level              confidence interval              rel. error

      90 %   1.005900 +/- 0.015530 = [0.990370, 1.021429]   0.015681
      95 %   1.005900 +/- 0.018559 = [0.987341, 1.024458]   0.018796
      98 %   1.005900 +/- 0.022116 = [0.983784, 1.028015]   0.022480
```

The confidence intervals are calculated for the confidence levels 90%, 95%, and 99%.

Calculating confidence intervals is complicated by the fact that many commonly collected statistics (e.g.., response times) are not independent. The algorithim used to calculate confidence intervals in CSIM19 groups the observations into batches, where the number of batches depends on the correlation found in the statistic. If a report is based on an insufficient number of batches, a message appears (instead of the calculated confidence intervals).

Run Length Control

CSIM19 provides a mechanism for running a model until a desired confidence level has been achieved for a specified statistic. However, it is possible that the model may require an excessive amount of computing time before the desired confidence level is achieved, so a maximum CPU time parameter is used to limit the execution time. The output report makes clear the terminating condition of the model.

The automatic run length control can be used with tables, qtables, meters, and boxes.

- *To declare, initialize, and use a table with run length control:*

```
main routine (sim):

const double CPU_TIME = 1 000.0;
const double CONF_LEVEL = 0.90;
const double ACCURACY = 0.01;

table *tbl;
extern “C” void sim()
{
        tbl = new table(“tbl”);
        tbl->run_length(ACCURACY, CONF_LEVEL, CPU_TIME);
        ...
        converged.wait()
        report();
}
```

Note: “Converged” is a built-in event that does not need to be declared or initialized.

Process Classes

In some models, it is convenient to be able to segregate different instances of a process (or processes) into classes for the purpose of reporting facility usage data (and possibly other statistics). Further information on this can be found in the CSIM Users' Guide.

Random Numbers and Streams

CSIM provides a set of functions that produce samples drawn from different probability distributions. These are all derived from a "random number generator". In the standard case, there is one random number generator function (one random number stream) used by all of the probability distributions. In some cases, it is convenient to have multiple streams of random numbers, so that each stream operates in a repeatable manner, even when the structure of the model is changed. The CSIM object "stream" serves this purpose.

The following example shows how random numbers and streams can be used:

- *To obtain a random number from the standard stream:*

```
const double SERVICE_TIME = 10.0   ;/*declare the mean service time */
float x;                           /*declare variables to contain random numbers */
...
...
x = exponential(SERVICE_TIME);     /* use standard random number stream with a negative
                                      exponential distribution on a mean of 10.0 */
```

- *To declare, initialize, and use a stream:*

```
const double SERVICE_TIME = 10.0;
stream *s
float x;
s = new stream();

x = s->exponential(SERVICE_TIME);
```

Successive streams are created with initial values (seeds) which are 100,080 values apart.

The seed of a stream can be changed by using the *reseed* function. The current value of the seed can be retrieved using the *stream_state* function.

CSIM19 includes the following built-in random number distribution functions:

```
uniform(min, max)
triangular(min, max, mode)
beta(min, max, shape1, shape2)
exponential(mean)
gamma(mean, stddev)
erlang(mean, var)
hyperx(mean, var)
weibull(shape, scale)
normal(mean, stddev)
lognormal(mean, stddev)
cauchy(alpha, beta)
hypoexponential(mean, var)
pareto(a)
zipf(n)
random_int(min, max)
bernoulli(prob_success)
binomial(prob_success, num_trials)
geometric(prob-success)
negative_binomial(success_num, prob_succes)
```

```
poisson(mean)
```

There is also a capability for generating random samples from an empirical distribution defined by a table.

- *To initialize and use randomly derived values specified as follows:*

value	**frequency**
1	.30
5	.60
9	.10

```
double prob[3] = {0.30, 0.60, 0.10};
double value[3] = {1.0, 5.0, 9.0};
double cutoff[4];
long alias[4];
...
setup_empirical(3, prob, cutoff, alias);
...
x = empirical(3, cutoff, alias, value);
```

Notice that the length of each of the auxiliary arrays, cutoff[] and alias[], is one greater of than the length of each of the parameter arrays, prob[] and value[].

Other Features

This tutorial has focused on the objects provided in the CSIM19 library. There are many other functions and procedures that help the CSIM user implement a simulation model. These include:

- Inspector functions that return the state of an object, the number of customers or messages waiting at the object and other important information about the object
- Status procedures (for most object types) that print a report on the status of all objects of that type. This can be useful in debugging a model.
- Report procedures which print summaries of statistics on the usage of facilities and storage as well summaries based on tables, histograms, qtables, and qhistograms
- More inspector functions that retrieve a number of items, including every item provided by CSIM reports, so that customized reports can be produced
- Routines to help with the management and execution of the model itself:
    - A procedure that resets all of the statistics being gathered, except for permanent tables which are never reset.
    - A "rerun" procedure that lets a model be destroyed and then rebuilt, possibly with some different features
    - A "debug" event trace can be "turned on", either by executing the "trace_on()" procedure or by providing an input argument when execution of the program is initiated

- Functions that allow the user to change the maximum number of objects that can be active in a model (the maximums are by object type)
- All of the files generated by the model (output, error and trace) can be directed to files under program control

A CSIM19 model can be embedded in another application. The user just has to provide a main routine that calls "sim".

All of the routines in the CSIM library have been optimized to support efficient execution of the model. All data structures are dynamically allocated, so there are no pre-determined limits to the sizes of models. The library does not have to be recompiled to handle large models.

Summary

CSIM19 has been designed to empower C++ programmers who need to build and use simulation models of complex systems. The programming approach to model building means that models of arbitrary levels of complexity and detail can be readily constructed and verified. CSIM19 does not embody any preconceived notions of how simulation models should be constructed (other than using the process-oriented paradigm).

Analysts and programmers interested in finding out more about CSIM19 and how it can be obtained should contact Mesquite Software, Inc.