Proceedings of the 1994 Winter Simulation Conference
ed. J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila

CSIM17: A SIMULATION MODEL-BUILDING TOOLKIT

Herb Schwetman

Mesquite Software, Inc.
8920 Business Park Drive
Austin, TX 78759, USA

ABSTRACT

CSIM is a simulation model-building toolkit that is used by C/C++ programmers to implement process-oriented, discrete-event simulation models. These models mimic the operation of complex systems, to give modelers insight into the dynamic behavior of these systems. Because CSIM models are C/C++ programs, there are virtually no limits on the level of detail, degree of complexity and size of the simulation models. Furthermore, CSIM uses special implementation techniques so that these models execute efficiently on a wide variety of systems platforms, including many UNIX workstations, PC's with Windows and Macintoshes.

This tutorial will introduce CSIM and then present some of the features that make CSIM a useful tool for building efficient simulation models of complex systems. Two examples will help illustrate these features. Integration of CSIM into other software products will also be discussed.

1 INTRODUCTION

CSIM is a library of routines (a toolkit) used by C or C++ programmers to write process-oriented simulation models. Because CSIM uses a "programming" approach to constructing simulation models, several benefits accrue:

- Programmers do not have to learn a new language; they begin immediately to build models using familiar techniques, and they have all of the software development tools of their programming environment available to them because they are using a standard programming language.
- There are almost no limitations on the style of model, model complexity and level of model detail; the models are programs and can make use of all of the power and flexibility of the underlying language.
- CSIM models can be executed on a wide variety of system platforms. CSIM users can develop their models on a PC or Macintosh and then execute them on a RISC workstation that could offer greater processing power. In addition, CSIM models can be easily sent to other users who have access to different platforms.
- CSIM models are compact and efficient. Because CSIM models are compiled C or C++ programs, they tend to be small and efficient. Furthermore, the CSIM toolkit has been written using dynamic memory allocation and specialized algorithms, to achieve both efficient memory utilization and good performance.

These factors mean that realistic models of complex systems can be implemented, debugged and deployed to help users investigate design trade-offs and achieve near-optimal operating configurations for these systems.

CSIM was developed in 1985 and over 300 copies have been sold to over 160 organizations world-wide. CSIM has been used to model many kinds of systems: including

- Computer systems and networks of computer systems,
- Software systems, including applications executing on multiprocessor systems,
- Communications systems,
- Air traffic control systems,
- An intelligent highway system,
- A satellite control system.

2 CSIM OVERVIEW

A CSIM model consists of resources and processes (entities) which compete for use of the resources. CSIM has the following kinds of resources:

- facilities,
- storages,
- mailboxes, and
- events.

In addition there are statements that deal with creating processes and with managing process interactions; these include the following:

- create - make a procedure into a process
- hold - allow simulated time to pass
- wait - cause a process to wait for an event to "occur"
- set - cause an event to be placed in the "occurred" state,

to name a few.

Finally, there are structures to help programmers construct useful models; these include:

- process_classes,
- tables and histograms,
- qtables and qhistograms, and
- random number functions and streams of random numbers.

In the C++ version, all of the resources and structures are classes, and all of the statements that deal with such a class are methods of that class. As an example, the following is a synopsis of the *facility* class:

```
// facility class declaration
class facility {
public:
  facility(char* name);      // constructor
  long reserve()             // reserve method
  void release()             // release method
  long qlength()             // cur. queue len.
  long status()              // BUSY or FREE
  float util()               // return utilization
  ...
  };
```

Thus, to construct a facility, the modeler writes:

```
facility cpu("cpu");
```

and to "use" this cpu (facility) for 10.0 units of simulated time:

```
cpu.reserve();
  hold(10.0);
cpu.release();
```

CSIM processes can use events to synchronize their activities. In the following segment, one process ("proca") waits for the other process ("procb"):

```
event go("go");
...
void proca()
{
  create("proca");
  go.wait();
  ...
}
```

```
void procb()
{
  create("procb");
  hold(1.0);
  go.set();
  ... }
```

It can be noted in this example that a CSIM process is just a C (or C++) procedure which executes a *create* statement. It is important to point out that when a *create* statement is executed, the newly created process is made "ready-to-execute" and the older process (the one that called the new process) continues execution.

3 FIRST EXAMPLE - A TELEPHONE HOTLINE

The following example illustrates some of the features of the CSIM toolkit. A company is setting up a customer service "hotline". They want to be certain that they will lose only a small percentage of the incoming calls. Some initial study has produced the following data:

Incoming call rate	4 calls/minute
Average call service time	5 minutes

The company is considering a telephone switch that operates as follows:

- an incoming call gets a busy signal if there are no lines available,
- if a line is available, then a call "rings" until a switch port is available,
- when a switch port is available, a recorded message is played and the call waits for an available operator,
- when the operator completes the call, it is terminated and tabulated as a "successful" call

At any point in this process before a call reaches an available operator, the caller can become discouraged and "hang up". As an initial guess, it is assumed that a caller will not tolerate a total delay of more than 75 seconds. The goal is to have at least 90 % of the incoming calls successfully completed.

A CSIM model of such a system can be described as follows:

- resources:
  - lines
  - switch_ports
  - operators
- processes
  - gen - call generator
  - call - models activities of a call

The process named "gen" is as follows;

```
/* call generator */
void gen()
{
  float inter_arrival;
  create("gen");
  inter_arrival = 1.0/call_rate;
  while(clock <= run_time) {
    num_act++;
    call();   /* initiate next call */
    hold(expntl(inter_arrival));
    }
}
```

This CSIM process generates incoming calls at exponentially distributed intervals to model the specified call rate.

The "call" process is first described as a simple CSIM process as follows:

```
/* call process */
call()
{
  int port_status, oper_status;
  float port_wait, operator_wait;

  create("call");
  num_calls++;
  if(avail(line) > 0) {
    allocate(1, line); /* line available */
    .../* call processing */
    deallocate(1, line);
    }
  else {
    busy_signals++; /* line not available */
    }
}
```

The set of incoming lines is modeled using a CSIM storage. The number of elements in the storage is the number of available lines. The *allocate* statement takes a line from the storage for use by this call; the *deallocate* statement returns a line to the storage when a call terminates.

The next step in the development of this model is to characterize calls contending for switch ports and for operators. If calls *could not* "hang up", then the following segment could be used:

```
/* call has line; now contends for port and oper. */
/* this segment does NOT model call "hang up" */

if(avail(line) > 0) {
  allocate(1, line);
    allocate(1, switch_port);
      hold(RECORDING_TM);
      reserve(operator);
        hold(expntl(SERV_TIME));
      release(operator);
      successful_call++;
    deallocate(1, switch_port);
  deallocate(1, line);
  }
else {
  busy_signals++;
  }
```

In this example, the lines and switch_ports are modeled as storages and the collection of operators as a multi-server facility.

In order to model callers becoming discouraged and "hanging up", a *max_wait* interval is defined. Then, an incoming call can either hang up while waiting for a switch port (the phone is still ringing) or while waiting for an operator ("your call will be handled in order; please stay on the line"). CSIM provides the *timed_allocate* and *timed_reserve* statements to model the case where a requester gives up if the request cannot be satisfied within a specified time limit. The above code segment is rewritten to use these statements:

```
/* call has line; now contends for port and oper. */
/* this segment models "hang up" */

if(avail(line) > 0) {
  allocate(1, line);
    total_wait = 75.0;
    port_wait = uniform(0.0, total_wait);
    port_status = timed_allocate(1, port,
            port_wait);
    if(port_status != TIMED_OUT) {
      operator_wait = total_wait - port_wait;
      oper_status = timed_reserve(operator,
            operator_wait);
      if(oper_status != TIMED_OUT) {
        hold(expntl(SERV_TIME));
        release(operator);
        successful_calls++;
        }
      else {
        operator_balks++;
        }
      deallocate(1, port);
      }
    else {
      port_balks++;
      }
    deallocate(1, line);
```

```
    }
  else {
    busy_signals++;
    }
```

The *timed_allocate* statement tests the specified storage; if there is a sufficient amount of available storage, the specified amount is removed and the process continues. If the amount of available storage is not sufficient to satisfy this request, then this process is *suspended* (deactivated) and placed on a queue of processes waiting for this storage to become available. Thus far, this procedure is identical to the normal *allocate* statement. However, with the *timed_allocate* statement, an additional parameter specifies the "time limit" for this request. If this time limit expires before the request for storage is granted, then the process is *resumed* (reactivated) with a status flag that indicates that the request "timed out". If the request is granted, a different status is returned. The *timed_reserve* operates in the same manner; the difference is that a *facility* is specified instead of a storage. These two statements, *timed_allocate* and *timed_reserve* allow the actions of discouraged callers to be modeled in a straightforward manner.

A version of this model was written using Borland Turbo C++ for Windows and run on a PC with a 486DX/33 processor. In one run, an eight hour day (28,800 seconds) was simulated. In this run, over 1900 calls were processed; the real time (CPU time) required was between three and four seconds. The results showed the following:

Call rate	4	calls/minute
Service time	5	minutes
Max wait time	75	seconds
Recording time	15	seconds
Number of lines	27	
Number of ports	24	
Number of operators	23	
Total calls	1923	
Successful calls	1772	
Percentage success	92.15	%

4 SECOND EXAMPLE - A CLIENT-SERVER COMPUTER SYSTEM

In this second example, an order processing system has a server (computer system) which accepts orders from client systems. Each incoming order is processed by updating several databases and printing invoices and pick lists. The clients are actually computer systems that scan order forms and submit each form as an order. The performance goal for this client-server system is to be able to "keep up" with an order stream of 2000 orders per hour.

This example was written in C++. The first step was to define two classes: an order class and an order_queue class, as follows:

```
class order_c {
protected:
  int number;
  TIME start_tm;
public:
  order_c(int n); { number = n; start_tm = clock;}
  TIME get_elapsed_time()
        {return clock - start_tm;}
};

class order_queue_c {
protected:
  mailbox *mb;
public:
  order_queue_c()    {mb = new mailbox("mb");}
  ~order_queue_c()   {delete mb;}
  void send(order_c* o) {mb->send((long) o);
  void receive(order_c** o)
        {mb->receive((long*)o); }
};
```

The order class is used in this example as just a "token" to pass between the client and the server. It does contain the start time for each order, so that the response time for each order can be tabulated.

The order_queue class includes a CSIM *mailbox*. A "message" (in this example an order) is placed in a mailbox (order_queue) by a *send* operation, and a message is removed from a mailbox by a *receive* operation. The order_queue class uses new *send* and *receive* methods, to handle order objects (as messages) correctly.

The "client" process just generates new orders at the specified rate, as follows:

```
void client()
{
  order_c *order; int order_number = 1;

  create("client");
  while(clock < SIM_TIME) {
    order = new order_c(order_number++);
    server_queue->send(order);
    hold(expntl(inter_arrival_time));
    }
  done->set();
}
```

The "server" process receives orders and then starts a sub-process (call "server_proc") to handle each incoming order, as follows:

```
void server()
{
  order_c *order;

  create("server");
  while(clock < SIM_TIME) {
    receive->server_queue(&order);
    server_proc(order);
    }
}
```

The process "server_proc" models the activities associated with processing each order. In this example, all of these activities are "modeled" by one service interval. In a more detailed model, all of the activities that are part of order processing could be modeled by many individual steps; these include computing (use the CPU facility), accessing different databases, and sending messages to other processes.

In this example, there is a restriction on the number of "server_proc" processes that can be "active" at the same time. The restriction on the number of simultaneously active "server_proc" processes is easily modeled with a *storage* of "server slots".

```
void server_proc(order_c *order)
{
  create("server_proc");
  server_slot->allocate(1);
    hold(expntl(SERVICE_TIME);
  server_slot->deallocate(1);
  resp_tm->record(order->get_elapsed_time());
}
```

The last statement in the listing for "server_proc" records (tabulates) the response time for each order in a table (named resp_tm).

There is a potential problem with this model as described above. If there are not enough server slots, then orders are not processed "fast enough" and the queue of unprocessed orders grows without bound. To alleviate this problem, the "client" process was modified, to check on the number of processes in the server_slot queue. If the number of queued process is 500, the model terminates, as follows:

```
void client()
{
  order_c *order; int order_number = 1;

  create("client");
  while(clock < SIM_TIME &&
        server_slot->qlength() < 500) {
    order = new order(order_number++);
    server_queue->send(order);
    hold(expntl(inter_arrival_time);
    }
  done->set();
}
```

The main ("sim") process that manages the execution of this model is as follows:

```
#include "cpp.h"

// define constants
// class definitions from above

order_queue *server_queue;
storage *server_slot;
table *resp_tm;
event *done;

extern "C" void sim()
{
  create("sim");
  init();        // initialize model
  server();      // start server
  client();      // start client
  done->wait();// wait for model to finish
  report();      // print CSIM report
}

void init()
{
  server_queue = new order_queue_c;
  server_slot = new storage("slots", N_SLOTS);
  resp_tm = new table('resp tm");
  done = new event("done");
}
```

The model was executed several times in order to determine the minimum number of slots required to "keep up". The parameter values were as follows:

Run time	8	hours
Mean order processing time	14	seconds
Order arrival rate	2000	orders/hour
Maximum queue length	500	

The results produced by this model are as follows:

Number of slots	Throughput	Response time
6	1554.9	500.5
8	1997.7	60.4
10	2031.6	16.9
12	1997.3	14.3

In the model with six slots, the run terminated early (at time 4160 seconds) because the maximum queue length value was exceeded. It can be seen that with eight or more slots the performance goal can be met.

5 INTEGRATION OF CSIM MODELS IN OTHER PROGRAMS

A major benefit of using a standard programming language to implement simulation models is that these models can be combined easily with other software components. For example, a simulation model of a system can be made part of a collection of software components used to design and evaluate large systems.

One application of CSIM was to embed a CSIM model in a product used to design and configure databases in large multiprocessor database management systems. The user is able to configure a database and a set of queries that will operate against that database. After the design is finished, a CSIM model of the system is used to estimate the performance of the queries operating on the database as configured on the host system. If the performance estimates from the simulation model are unacceptable, the database and/or the underlying system can be modified and the performance estimated again.

In another application, a CSIM model was embedded in a system that models satellite orbits and presents the results in graphical form. The entire system was used to train people in the operation of the satellite network.

Recently, the CSIM model of the telephone bank described above was embedded in a Windows application, so that the user could use a "point and click" interface to change the input parameters and view the results in graph form.

In each of these examples, a CSIM model was embedded in a larger application. The embedding took advantage of the fact that the model and the library are all C or C++ programs. In addition, CSIM has many functions and procedures that give the programmer access to the procedure-level interface for every feature of CSIM. As an example, while the *report* procedure gives a "standard" report on the use of all of the simulated resources, every item in this report can be accessed individually by a CSIM function. In addition, while the normal style of developing a CSIM model is to use the CSIM-provided *main* procedure, the programmer has the option of providing a different (customized) *main* procedure when this is necessary. Also, the library routines that create CSIM resources guarantee that the CSIM runtime environment is initialized before the resources are actually required.

6 SUMMARY

CSIM helps programmers construct simulation models of systems. Because the models are written in a standard programming language (C or C++), CSIM is a "quick start" for most programmers. One programmer reported success in writing and executing a "first" model within 30 minutes of installing the toolkit.

The process-oriented approach embodied in CSIM is a natural way of expressing the behavioral aspects of many kinds of systems. Most systems can be easily described in terms of resources and processes competing for use of these resources.

Since 1986, over 160 organizations have acquired the CSIM toolkit. Many of these have used CSIM to develop models of *their* systems. Some of them have embedded these models in other software products. These analysts and developers have found CSIM to be an inexpensive and convenient tool for implementing simulation models. The flexibility, efficiency and portability of these CSIM models allow them to be deployed where they are needed. CSIM has helped these organizations meet their needs for simulation models.

CSIM17 is now available from Mesquite Software, Inc.

ACKNOWLEDGMENTS

CSIM is copyrighted by Microelectronics and Computer Technology Corporation (MCC). CSIM17 is supported and marketed by Mesquite Software, Inc. under a license from MCC.

REFERENCES

Edwards. G and R. Sankar. 1992, Modeling and simulation of networks using CSIM. *Simulation* 58:2, 131-136.

Schwetman, H. 1990. Introduction to process-oriented simulation and CSIM. In *Proceedings of the 1990 Winter Simulation Conference*, ed. O. Balci, R. Sadowski, and R. Nance.

Schwetman, H. 1988. Using CSIM to model complex systems. In *Proceedings of the 1988 Winter Simulation Conference*, ed. M. Abrams, P. Haigh, and J. Comfort.

Schwetman, H. 1986. CSIM: A C-based, process-oriented simulation language. In *Proceedings of the 1986 Winter Simulation Conference*, ed. J. Wilson, J. Henriksen, and S. Roberts

AUTHOR BIOGRAPHY

HERB SCHWETMAN is founder and president of Mesquite Software, Inc. Prior to founding Mesquite Software in 1994, he was a Senior Member of the Technical Staff at MCC from 1984 until 1994. From 1972 until 1984, he was a member of the staff of the Department of Computer Sciences at Purdue University. He received his Ph.D. in Computer Science from The University of Texas at Austin in 1970. He has been involved in research into system modeling and simulation as applied to computer systems since 1968.