Proceedings of the 1986 Winter Simulation Conference
J. Wilson, J. Henriksen, S. Roberts (eds.)

CSIM†: A C-BASED, PROCESS-ORIENTED SIMULATION LANGUAGE

Herb Schwetman
Microelectronics and Computer Technology Corporation
P.O. Box 200195
Austin, TX 78759, USA

ABSTRACT

CSIM is a process-oriented simulation language which is implemented as a superset of the C programming language. Using CSIM, a simulation programmer is able to quickly construct concise models of systems and then to execute these models in an efficient manner. In addition to supporting process-oriented simulation, CSIM as a number of additional features dealing with modeling system resources, message passing, data collection and debugging which ease the task of building simulation models.

1. INTRODUCTION

Computer-based simulation models have become important tools in the analysis of complex systems. A simulation modeling methodology has emerged as the use of these models has increased. This methodology is based on decomposing system behavior into a sequence of discrete events, where an event corresponds to a change of state of the system (Law and Kelton 1982). These state changes occur instantaneously, so events take no time; simulated time passes between events.

Software packages and programming languages specialized toward implementing computer simulation models have been developed. These can dramatically decrease the time and effort required to implement, debug and utilize these models. CSIM is one such language. It uses a process-oriented view of a system, in contrast to an event-oriented view which is common in many other languages.

CSIM is a process-oriented simulation language which is implemented on top of the C Programming Language (Kernighan and Ritchie 1978). The current version of CSIM runs on the UNIX†† operating system (the 4.2 BSD Version) on several computer systems including the DEC VAX and the SUN III workstation. CSIM gives the simulation programmer facilities for defining processes, for initiating these processes, and for synchronizing these processes. Programs written in CSIM are really C programs, with calls to procedures which form an extended run-time support system. Thus, users have all of the power and convenience of the C language, with the additional feature of being able to create process-oriented simulation models.

The primary purpose of this paper is to present CSIM as a tool, useful for easily modeling a variety of systems. In order to provide a basis for understanding CSIM, the first two sections introduce discrete event simulation, as well briefly describing both event-oriented and process-oriented approaches to simulation. Several examples illustrate the basic features of CSIM.

2. APPROACHES TO SIMULATION

There are three different approaches to discrete event simulation (Nance 1981): event-oriented simulation, activity-oriented simulation, and process-oriented simulation (Franta 1977) (Pritsker and Pegden 1979) (Saydam 1985). In the event-oriented approach, the simulation programmer defines events and then writes routines which are invoked as each kind of event occurs. Simulated time may pass between the events. In the activity-oriented approach, the programmer defines activities which are started when certain conditions are satisfied. In many cases, this type of simulation uses a simulated clock which advances in constant increments of time. With each advance, a list of activities is scanned, and those which have become eligible are started. This type of model is used more often with simulating physical devices and will not be discussed futher in this paper. In the process-oriented approach, the programmer defines the processes (entities, transactions, etc.) which "use" the resources of the system. Each process can be in one of three states: active (currently being processed by the simulator), holding (waiting for an interval of simulated time to pass) or waiting (in a queue) for an event to occur. Simulated time passes only when processes are in the hold state.

Event-oriented simulation is probably the more prevalent approach in current use, for two reasons: event-oriented simulation models can be implemented in standard high-level programming languages (e.g., Fortran, Pascal and C) and a number of packages and languages which aid the development of these kinds of models are widely available (e.g., Simscript (Kiviat, Vilaneuva and Markovitz 1975) and GASP-IV (Pritsker 1974)).

Process-oriented simulation models are usually implemented in special simulation languages which support this approach. Some notable examples are GPSS (Gordon 1978), Simula (Dahl and Mygaard 1967), SLAM (Pritsker and Pegden 1979) and ASPOL (MacDougall and McApline 1973, MacDougall 1974, 1975, 1976). Of these, ASPOL and Simula are general purpose programming languages (enhanced with the addition of some simulation features), while SLAM and GPSS are special purpose languages, tailored to implementing simulation models. In the next

†† UNIX is a trademark of AT&T Bell Laboratories.

section, we will discuss the features which must be present to support process-oriented simulation. We then present a new process-oriented simulation language, CSIM, which has been implemented.

The choice of which approach to use is usually limited to the choice of languages and packages available on a specific system. This is unfortunate, as there are situations in which the nature of the model to be developed might make one approach preferable to the other. The choice is in some sense a matter of taste, as any model developed using one approach could have been implemented using the other. However, some models are more straightforward when implemented using a particular approach.

As an example, consider the simple M/M/1 single server queue. In this system, entities (transactions, jobs, customers, etc.) arrive at the queue at intervals called arrival intervals, wait for the single server to become available, use this server for a period of time (the service interval) and depart. There is an infinite (well anyway large) customer population. In the M/M/1 version of this simple queueing system, both the arrival intervals and the service intervals are exponentially distributed. To simulate this system using the event-oriented approach, the programmer could devise a model with two events:

- the arrival event, and
- the departure event.

Once these two events are defined, the processing associated with each event is as follows:

- when an arrival event occurs, generate the next arrival event to occur in the future after an arrival interval; check the server, to see if it is busy or free; if busy, put this arrival on a queue of waiting tasks; if the server is free, assign this arrival to the server and generate a departure event to occur in the future after a service interval.
- when a departure event occurs, put the server back in the free state and cause the completed entity to depart; check the queue of waiting requests; if a request is waiting, get the next one from the queue, assign it to the server and schedule a departure event.

In the process-oriented approach, the programmer could devise a model with two independent types of processes:

- the arrival-generator process, and
- the processes which represent the entities which are being processed by the simulated system.

The activities of these types of processes can be described as follows:

- the generator process initiates the next arriving entity (process), and then allows an arrival interval to pass before initiating the next arriving entity; and
- the entity process requests use of the server; when the entity process receives control of the server, it does a hold for a service interval (this cause the simulator to simulate the passage of the specified amount of time), releases control of the server, and terminates.

The runtime environment for a process-oriented simulator must include facilities for managing these processes and for implementing statements such as reserve, hold and release.

The point of this example (with two implementations) is to demonstrate that in the event-oriented approach, the programmer defines events and event processing procedures. In the process-oriented approach, the programmer defines the necessary interacting processes; the focus is on the entities and descriptions of their behavior. In many cases, this latter approach corresponds more nearly to the system being modeled. In fact, when only event-oriented simulation is available for use, it is common practice to design the model in terms of the interacting processes and then to (manually) decompose the model into the underlying events, for use in the event-oriented simulation model.

In a process-oriented simulator, the simulator automatically maps the process-oriented model onto the underlying event-oriented model. Thus, process-oriented simulation requires a more complex run-time environment, to support simulation of multiple interacting processes executing on a single processor computer. Such environments have been developed and are available for some computer systems.

3. PROCESS-ORIENTED SIMULATION

As stated above, in a process-oriented simulator, the programmer defines the model in terms of interacting processes. In this context, we define a process as a complete computing activity. In a process-oriented simulator, a process is an "independent" program or procedure which can execute "in parallel" with other processes. The notion of "in parallel" is used with some liberty, because, on a single cpu computer, the processes only appear to execute in parallel: they simulate parallel or simultaneous sequences of activities.

It is also possible to define processes to simulate the resources of a simulated system. This would be a resource-oriented view of the system, in contrast to the transaction-oriented view described above. In any process-oriented model, deciding just what will be the processes and what will be the resources is a critical decision. In most of what follows, we will use the transaction-oriented approach, but, in many cases, this choice is one of taste.

Many kinds of systems can be modeled as collections of simultaneously active processes. In the example above (the single server queue), the arriving entities could be modeled easily by one process, which is initiated (a new instance of the process is activated) whenever an arrival occurs. Examples of other simulation models which can use this approach include:

- models of computer systems, where jobs or transactions being processed by the system are modeled as processes,
- transportation models, where vehicles are modeled as processes,
- models of commercial establishments, where customers are modeled as processes, and
- models of manufacturing operations, where component assemblies are modeled as processes.

At the user level, we need to be able to define and describe processes, and then to be able to initiate processes at specified points in simulated time. Each of the initiated processes must be able to execute simultaneously with other

processes (or at least appear to do so). This means that each process must have a "private" data area, to store results of computations and to save data values (such as the time of process generation). All process-oriented simulation systems have these facilities.

Furthermore, it is necessary for the simultaneously active processes to interact with each other. In the M/M/1 example, each process had to be able to acquire exclusive control of the single server. Thus, process-oriented simulation systems have several mechanisms which allow processes to communicate and synchronize with each other. Different systems do this in different ways. Most systems have the equivalent of a facility (or server) which can be reserved for exclusive use by individual processes. In addition, the system must properly handle a process which requests use of a facility which is already busy. The most common solution is to automatically suspend the requesting process and to place it on a queue of processes waiting to use the facility. As each requester gains control of the facility, it is reactivated (allowed to resume). At some point, that process will release the facility (give up control of the facility); when this occurs, the next waiting process can be given access to the facility, and so on.

To summarize, process-oriented simulation allows simulation programmers to model systems by defining interacting processes as abstractions of the active entities in the system. Each process must be able to execute in parallel with other active processes; each process requires a private data area; and active processes must be able to interact (communicate and synchronize) with other active processes. The underlying support system provides these facilities. In particular, the underlying system must be able to manage conflicting requests for exclusive use of simulated resources. The next section describes an implementation of process-oriented simulator which provides these features (and more).

4. CSIM

CSIM provides an extended set of features which facilitate the implementation of process-oriented simulation models. These are implemented as a set of extensions to the C programming language. A CSIM program accesses these features via function or procedure calls from a C program, very similar to the way GASP features are accessed from FORTRAN programs. The major difference is that CSIM provides a process-oriented view, while GASP provides the event-oriented view.

The CSIM programmer is able to create concise models (see the following examples) which are much shorter than an equivalent program written using only C constructs. Also important in an evaluation of CSIM is the fact that, because CSIM is based on C, all of the C programming environment is available to the CSIM programmer. A CSIM program can perform useful computation as well as being a simulation model of some system. This latter point is one of the major motivations for using CSIM: actual C procedures can be part of the CSIM model. Another motivation is that CSIM is available under UNIX. The conciseness of CSIM programs, the variety of features available and the fact that a CSIM model is a compiled program (as opposed to being interpreted) means that most process-oriented models can be implemented quickly and executed efficiently.

All of the features of CSIM are described in a reference manual which is available from the author. In the next sections, some of these features will be presented via a set of simple CSIM programs. It is assumed that the reader has a knowledge of C.

The first example is the simple M/M/1 queue, cited earlier. The listing for this program is shown in Figure 1. The output for this example is shown in Figure 2.

```
/* simulate an M/M/1 queue */

#include "csimlib/csim.h"

#define SVTM      1.0
#define IATM      2.0
#define NARS      5000

int f, done;
int cnt;

sim()
{
      int i;

      create("sim");

      f = facility("queue");
      done = event("done");

      cnt = NARS;
      for(i = 1; i <= NARS; i++) {
            hold(expntl(IATM));
            cust();
            }
      wait(done);
      report();
}

cust()
{
      create("cust");

      reserve(f);
            hold(expntl(SVTM));
      release(f);
      cnt--;
      if(cnt == 0)
            set(done);
}
```

Figure 1: Example #1 - M/M/1 Queue

Example #1 illustrates some of the most basic concepts of process-oriented simulation. The two process descriptions (sim and cust) look like C procedures. The appearance of the *create* statement causes each of them to be instantiated as processes every time the *create* statement is executed. Thus the first process (it must be named sim) creates itself as a process, does some housekeeping (see below) and then enters a for-loop which repeatedly executes a *hold* for the next interarrival period and then generates the next arrival. The interarrival periods are randomly drawn from an exponential distribution with mean specified by the constant IATM (1.0 in this example). After the last arrival is generated, the process sim executes a *wait* statement; the process cust is designed to send a signal when the last arrival has departed. Thus, this *wait* causes sim to wait for the last

```
Tue Jul  2 14:58:44   CSIM Simulation Report Version 8

Model: CSIM        Time:          10041.661
                   Interval:      10041.661
                   CPU Time:         39.867 (seconds)

                        Facility Usage Statistics

+--------------------+-------------means---------------+---counts----+
facility      srv    disp  serv_tm  util    tput   qlen   resp    cmp     pre

queue                      0.992   0.494     0.5  0.991  1.989   5000       0
```

Figure 2: Example #1 - Sample Output

arrival to leave; when this happens, the *report* statement causes a summary report to be printed; the simulation then terminates (when the process sim ends).

The two "housekeeping" statements mentioned above declare two simulation data objects: f is a pointer to a *facility* and done is a pointer to an *event* (here, an event is a data object which will be defined, not an event in the execution of the model). A facility is a data object which can be *reserved* and *released* by processes. It is used to model the single server queue described in this example. A facility can be in one of two states: BUSY or FREE. If a process *reserves* a FREE facility, the facility is assigned to that process and the process continues. If a process *reserves* a BUSY facility, the process is suspended until another process *releases* that facility. When this happens, the waiting process is given control of the facility and execution resumes.

An *event*, another kind of data object, also has two states: OCCURRED and NOT-OCCURRED. When a process *waits* for an event in the OCCURRED state, the event is automatically placed in the NOT-OCCURRED state and the process continues. When a process *waits* for an event in the NOT-OCCURRED state, execution of the process is suspended. When some other process *sets* that event, the event is placed in the OCCURRED state and all waiting processes are placed back in the active state; as this happens, the event is automatically returned to the NOT-OCCURRED state.

In the example, the multiple instances of the cust process compete for use of the queue. Each time cust is initiated (by the sim process), the *create* statement causes a new instance of cust (the process) to be created and allows the initiator (sim) to continue. Each instance of cust tries to gain access to the queue by executing the *reserve* statement. Because each instance of a process inherits its process priority from the initiating process, all instances of cust have the same priority. The processes waiting to use the queue are selected in order by priority; since, in this example, they all have the same priority, the order of selection becomes first-come, first-served. When an instance of cust does gain access to the queue, it executes a *hold* statement, which allows simulated time to pass. After this, the process *releases* the queue, allowing the next waiting process to gain control. As each process departs, it decrements a counter; the last process to depart executes a *set* statement, which allows sim to resume.

The output for Example #1 gives an overview of the usage of the resources of the simulated system (the queue). The performance parameters produced by this run include the mean service time, the utilization of the queue (percentage of elapsed time busy), the throughput rate (customers per unit time), the mean queue length (average number of customers either waiting or in service) and the response time for a customer (time of arrival to time of departure). In this example, 5000 customers completed service at the queue and the model ran for 10,041.661 simulated units of time. On the VAX 11/750 used to execute this program, 39.867 seconds of user-mode cpu time were required.

This example has illustrated the fundamental concepts which are present in a process-oriented simulation model, namely describing and initiating processes and process interaction, as typified by the declaration and use of facilities and events in CSIM. It is easy to see how more complex systems could be modeled by extending this basic single resource model. For example, a system in which customers visit a number of queues could be modeled by adding more facilities and then having the customer visit (*reserve, hold* and *release*) each of these in some specified pattern. Another extension could involve the use of multi-server facilities (e.g. having two equally competent tellers in a bank, with a single common queue of waiting customers). The *facility* statement can have a second argument which specifies the number of servers (the default is one).

5. SIMULATING COMPUTER SYSTEMS

A major use of process-oriented simulation has been to simulate the operation of computer systems and system components. Simulation-based studies have focused on analyzing system performance, estimating program performance, predicting subsystem performance (e.g. and I/O subsystem) and verifying component level operation and performance.

CSIM has several features which are particularly useful in this type of application. Example #2 (see Figure 3) illustrates a number of these. The output generated by an execution of this model is shown in Figure 4.

In this example, some of the resources of the system are represented as facilities - the cpu and the disk drives. The cpu is declared to be a two-server facility; the disk drives are a set of four, single-server facilities. The main memory is declared using the *storage* statement. A storage is similar

to a facility, except that a process can *allocate* a specified amount of storage. The *allocate* compares the amount requested with the amount available in the storage. If the amount available is sufficient, it is decreased by the amount requested and the requesting process continues. If the amount requested exceeds the amount available, the requesting process is suspended and placed on a queue (in order by priority) for that storage. When enough storage has been *deallocated*, the waiting request can be satisfied and the process allowed to continue.

```
/* CSIM Example of General Computer System */

#include "csimlib/csim.h"

#define NJOBS     100
#define NCPUS     2
#define NDISKS    4
#define AMTMEM    20

#define INTARV    1.0
#define MNCPU     0.25
#define MNDSK     0.030

#define TOTCPU    1.0
#define EPS 0.0005

int cpu,
    disk[NDISKS],
    mem,
    done,
    act;

sim()
{
      int i;

      create("sim");

      cpu = facility("cpu", NCPUS, pre_res);
      facility_set(disk, "disk", NDISKS);
      mem = storage("memory", AMTMEM);
      done = event("done");

      act = NJOBS;
      for(i = 1; i <= NJOBS; i++) {
            job(i);
            hold(expntl(INTARV));
            }
      wait(done);
      report();
}

job(i)
int i;
{
      int j, amt, iodone;
      float t, cpt, x;

      create("job");
      set_priority(i);
      iodone = event("iodone");

      t = clock;
      amt = random(5, AMTMEM);
      allocate(amt, mem);
            cpt = erlang(TOTCPU, 0.5*TOTCPU);
            while(cpt > EPS) {
                  j = random(0, NDISKS-1);
                  io(j, iodone);
                  x = hyperx(MNCPU, 4.0*MNCPU);
                  cpt -= x;
                  use(cpu,x);
                  wait(iodone);
            }
      deallocate(amt, mem);
      act--;
      if(act == 0) set(done);
}

io(d, ev)
int d, ev;
{
      create("io");

      reserve(disk[d]);
            hold(expntl(MNDSK));
      release(disk[d]);
      set(ev);
}
```

Figure 3: Example #2 - Computer System Model

The cpu in this example is defined by a form of the *facility* statement which allows the programmer to specify a scheduling strategy (discipline) for that facility. In the example, a preempt-resume strategy based on process priorities is specified. In order to utilize this feature, the programmer specifies a *use* (instead of the the *request, hold, release* sequence which is normally given). The *use* statement is interpreted to mean that the specified facility is to be used for a period of time, and this use is governed by the scheduling policy in effect for the facility. Five different scheduling policies are provided in CSIM.

This example also uses some of the different (pseudo-) random sample generators provided in CSIM. The exponential distribution (the *expntl* function) was called to generate the interarrival and disk service intervals. In addition, the Erlang distribution (the *erlang* function), the hyperexponential distribution (the *hyperx* function) and the uniform distribution (the *random* function) were used. Seven different random sample generators (functions) are part of CSIM.

6. SIMULATED PARALLEL PROCESSING

The fact that CSIM is a programming language means that it is possible to write programs which implement parallel algorithms and then to estimate the performance of these algorithms on various multiprocessor systems. The technique which is used to simulate execution of a parallel algorithm using CSIM is to implement the algorithm with the parallel segments of the algorithm written as separate processes. As each parallel segment of the algorithm begins, a processor is *reserved*. After receiving a processor, the segment gets the current user-mode cpu time (from the operating system's clock). When the parallel segment completes, the user-mode cpu time is obtained again. The difference between these two clock readings is the elapsed processor time required for that segment of the computation. The segment executes a *hold* for this amount of time and then *releases* the simulated processor. This technique can provide a rough approximation to the time required to execute the algorithm on a parallel system.

Example #3 (see Figure 5) demonstrates use of this technique on a simple parallel algorithm (to add up the elements of a vector of integers). In Example #3, the execution of this algorithm is simulated on a series of systems, each with a number of processors and a common block of main memory. The output from this example is given in Figure 6.

```
Wed Jul  3 11:26:12   CSIM Simulation Report Version 8

Model: CSIM       Time:          205.517
                  Interval:      205.517
                  CPU Time:        8.383 (seconds)

                    Facility Usage Statistics

+--------------------+-------------means---------------+---counts----+
facility     srv    disp  serv_tm   util   tput   qlen   resp    cmp    pre

cpu            0 pre_res   0.301  0.910    3.0                  622
cpu            1 pre_res   0.302  0.179    0.6                  122
cpu              pre_res   0.301  1.090    3.6  1.091  0.301    744      4
disk[0]                    0.028  0.025    0.9  0.025  0.028    183      0
disk[1]                    0.029  0.025    0.8  0.025  0.030    174      0
disk[2]                    0.031  0.029    0.9  0.029  0.032    191      0
disk[3]                    0.035  0.033    0.9  0.033  0.035    192      0

                    Storage Usage Statistics

+---------------+-----------------means--------------+---counts----+
storage   capacity     amt     util  srv_tm    qlen     resp     cmp     que

memory          20    12.8    0.712   2.293  26.030   53.495     100      82
```

Figure 4: Example #2 - Sample Output

Example #3, in addition to illustrating a method for simulating execution of a parallel algorithm on a multiprocessor system, also demonstrates use of the *rerun* statement. In the program, the model is executed several times; each time, there is one additional processor in the simulated system. The *rerun* statement causes the simulation support system to be completely reinitialized (with the exception of the random number function). This feature can be used to obtain several independent executions of the model.

The method used to simulate execution of a program on a multiprocessor system has a couple of flaws. One is that there was no attempt made to model the effects of contention for access to shared memory. In real systems, it is common for the execution times of the parallel segments to be lengthened (slightly) as more processors are simultaneously accessing main memory. There are ways of modeling these effects, but they were not implemented in this example. Another problem can be seen in the output for this example: no process ever had to wait to obtain the lock (for the global variable named sum). A study of the underlying simulation mechanisms reveals that no process will ever wait for access to this lock. The problem is that no (simulated) time passes while the lock is being held. In a real system, time would pass while the lock was held, creating the potential for delays in accessing the lock. The simulation model could be corrected to allow for these delays, but this would require a overt recognition of these problems by the programmer.

7. MESSAGES AND MAILBOXES

A common technique for interprocess communication is to use messages and mailboxes. CSIM provides a simulation data object called a *mailbox*. By using mailboxes, a process can send a message, consisting of a single integer, to another process. Because C can represent a pointer to any arbitrary data structure as an integer, passing a single integer as a message is equivalent to passing a data structure. Of course, a process cannot pass a pointer to a data structure within its "private" data area, as this area will not be locatable when another process is in the active state (this is because of the way processes and private data areas are managed by the CSIM support system).

A mailbox is created by a process. A process can send a message to a mailbox (using the *send_msg* function), and it can receive a message from a mailbox (*receive_msg*). The mailbox contains a queue of unreceived messages. When a process does a *receive_msg*, specifying a particular mailbox, it either gets the next unreceived message or it enters the wait state. The mailbox also contains a queue of processes waiting to receive messages. Whenever a message arrives at a mailbox, the process at the head of this queue (if there is one) is reactivated and the message is passed to that process. A process can determine the number of unreceived messages in a mailbox.

Example #4 (see Figure 7) is a CSIM program which makes use of messages and mailboxes. The program prints a list of all of the prime numbers less than or equal to a specified number. The technique used is a variation on the famous Sieve of Eratosthenes. In the CSIM version, the first process (sim) generates a stream consisting of all of the integers less than or equal to the specified limit. Sim also initiates a process (called proc) which will receive the stream of integers via a mailbox. As proc receives its input stream, the first number in the stream is the next prime number in the stream. Each instance of proc initiates another instance of itself. It then checks its stream of input numbers by doing a modulus operation (determine the remainder) with its prime number. All number which are not multiples of this prime are sent to the successor process. The output for this example is in Figure 8.

```
/* Sum Elements of a Vector in Parallel */

#include "csimlib/csim.h"

#define     LENGTH      100000
#define     N           5

int cpu,
    lock,
    done;

int a[LENGTH],
    sum,
    act;
float t1,
      t2,
      etime[N];
sim()
{
      int irun, i;


      init();
      for(irun = 1; irun <= N; irun++) {
            create("sum");
            cpu = facility("cpu",irun);
            lock = facility("lock");
            done = event("done");
            t1 = clock;
            act = irun;
            sum = 0;
            for(i = 1; i <= irun; i++)
                  add(irun, i);
            wait(done);
            t2 = clock;
            etime[irun] = t2 - t1;
      printf("LENGTH = %d, NPROC = %d, sum = %d0,
                  LENGTH, irun, sum);
      printf("irun = %d, etime[irun] = %.3f0,
                  irun, etime[irun]);
            report();
            rerun();
            }
      prtres();
}

add(nproc, i)
int nproc, i;
{
      int j, k1, k2, l, lsum;
      float x1, x2;

      create("add");
      reserve(cpu);
            x1 = cputime();
            lsum = 0;
            j = LENGTH/nproc;
            k1 = (i-1)*j;
            k2 = (i != nproc) ? i*j : LENGTH;
      printf("process %d, k1 = %d, k2 = %d0,
                  i, k1, k2);
            for(l = k1; l < k2; l++)
                  lsum += a[l];
            reserve(lock);
                  sum += lsum;
            release(lock);
            x2 = cputime();
            hold(x2 - x1);
            act--;
            if(act == 0)
                  set(done);
      release(cpu);
}
```

```
init()
{
      int i;

      for(i = 0; i < LENGTH; i++)
            a[i] = 1;
}

prtres()
{
      int i;

      printf("0);
printf("Num. ProcsSim Elapsed Time (seconds)0);
      for(i = 1; i <= N; i++)
            printf("%2d%.3f0,
                  i, etime[i]);
}
```

Figure 5: Example #3 - Simulated Parallel Processing

8. DATA COLLECTION AND DEBUGGING

An important feature in any simulation package is the set of features for collecting data on the performance of the simulated system. The examples above have demonstrated the "automatic" features available in CSIM (the reports on the usage of facilities and storages), as well as the "do-it-yourself" approach (as in Example #4). An important third technique for data collection makes use of tables and (numerical) histograms. In CSIM, there are simulation data objects which can be created and then used by the programmer to collect data in some specific forms. These are described in this section.

There are two basic simulation objects which can be used to collect data: one is based on recording values in a *table* or a frequency *histogram* and the other is based on collecting information about the state of a resource, e.g., the queue length distribution, where the state is the number of customers at a resource.

With the first approach, the CSIM statements *table* and *histogram* are used to create either a table or a table with a frequency histogram appended to it. The *record* statement is used to record floating point values in these simulation objects. Example #5 (see Figures 9 and 10) demonstrates the use of these statements. In this example, the program generates 1,000 random samples drawn from an exponential distribution and records them in the table. The output consists of both a statistical summary of these values (mean, variance, minimum and maximum) and a frequency histogram of the recorded values.

The other form of data collection uses *qtables* and *qhistograms* to tabulate data on states of the system or system components. One example would be to tabulate changes in the length of a queue as the model progresses. The *note_entry* and *note_exit* statements would then be used to notify the data collection routines of changes in the queue length.

```
Tue Jul  9 08:58:37   CSIM Simulation Report Version 8


Model: CSIM      Time:            0.533
                 Interval:        0.533
                 CPU Time:       16.550 (seconds)


                        Facility Usage Statistics

+--------------------+-------------means---------------+---counts----+
facility     srv    disp  serv_tm  util   tput   qlen   resp    cmp    pre

cpu            0          0.517  0.969   1.9                     1
cpu            1          0.533  1.000   1.9                     1
cpu            2          0.517  0.969   1.9                     1
cpu            3          0.533  1.000   1.9                     1
cpu            4          0.517  0.969   1.9                     1
cpu                       0.523  4.906   9.4   4.906  0.523      5      0
lock                      0.000  0.000   9.4   0.000  0.000      5      0

      Number of Processors      Simulated Elapsed Time (seconds)

                1                        2.583
                2                        1.350
                3                        0.867
                4                        0.667
                5                        0.533
```

Figure 6: Example #3 - Output

```
/* the Sieve of Eratothenes */

#include "csimlib/csim.h"

#define     N     500
#define MAXLINE 9

int cpu,
    done,
    act,
    linect;

sim()
{
      int i, mb;

      create("sim");

      cpu = facility_ms("cpu", N/3);
      done = event("done");

      printf("The primes <= %d are:0, N);
      linect = MAXLINE + 1;

      act = 0;
      mb = mailbox();
         reserve(cpu);
            proc(mb);
            for(i = 2; i <= N; i++) {
                  send(mb, i);
                  hold(1.0);
            }
      send(mb, -1);
      wait(done);
      release(cpu);
      if(linect > 0)
            printf("0);
      printf("0cpu time = %.3f seconds0,
                  cputime());
}
```

```
proc(mbox)
int mbox;
{
      int my_prime, next_mb, num;

      create("proc");
      act++;
      reserve(cpu);
            receive(mbox, &my_prime);
            if(my_prime >= 0) {
                  if(linect > MAXLINE) {
                        printf("0);
                        linect = 0;

                        }
                  printf("%8d", my_prime);
                  linect++;
                  next_mb = mailbox();
                  proc(next_mb);
                  do {
                        receive(mbox, &num);
                        if((num % my_prime) != 0)
                              send(next_mb, num);
                  } while (num >= 0);
                  send(next_mb, -1);
            }
      release(cpu);
      act--;
      if(act <= 0)
            set(done);
}
```

Figure 7: Example #4 - Sieve of Eratosthenes with Mailboxes

```
The primes <= 500 are:

  2   3   5   7  11  13  17  19  23  29
 31  37  41  43  47  53  59  61  67  71
 73  79  83  89  97 101 103 107 109 113
127 131 137 139 149 151 157 163 167 173
179 181 191 193 197 199 211 223 227 229
233 239 241 251 257 263 269 271 277 281
283 293 307 311 313 317 331 337 347 349
353 359 367 373 379 383 389 397 401 409
419 421 431 433 439 443 449 457 461 463
467 479 487 491 499

cpu time = 14.333 seconds
```

Figure 8: Example #4 - Sample Output

```
/* Data Collection */

#include "csimlib/csim.h"

#define N   10
#define MN  2.0

int hist;

sim()
{
      int i;

      create("sim");

      hist = histogram("expntl", N, 0.0, 4*MN);
      for(i = 1; i <= 1000; i++)
            record(expntl(MN), hist);
      report();
}
```

Figure 9: Example #5 - Collecting Data

The major debugging aid in CSIM is a *trace*, which can be turned on or off either under program control using the *trace_on* and *trace_off* statements or when the program is invoked using the -tr option on the invoking command line. The trace output is a fairly detailed listing of the activities of the processes of the model. The level of detail is usually sufficient to allow most bugs to be detected and corrected. In addition, there are many checks for errors in the runtime support system. These checks detect illegal conditions and print an error message before halting.

9. SUMMARY

CSIM is a process-oriented simulation language, in many respects similar to ASPOL. Because it is a complete programming language (it is embedded in C), it can be used in a variety of simulation projects. In addition to the usual system simulation models, CSIM can also be used to simulate the execution of programs on systems with different architectures, as was illustrated in Example #3. The process-oriented nature of CSIM means that many types of models can be easily implemented.

CSIM is currently implemented on a VAX computer system using the UNIX operating system. The implementation is really a set of functions and procedures which the model (the C program) calls. The *create* function causes a procedure to be set up as a process, capable of executing in a pseudo-parallel manner with other processes. The run-time support package includes a small number of short routines written in VAX Assembly Language; these routines manipulate the C run-time stack and save and restore some registers and jump addresses.

CSIM is a useful simulation tool. Several large projects have used CSIM to model various aspects of program and system behavior. The simple examples found in this paper illustrate the rich variety of features which enable a simulation programmer to easily develop, test and use system models.

Because there is currently no CSIM compiler, there is no good mechanism for checking that a CSIM program is syntactically correct: it is possible to write a program which compiles, but which has CSIM errors. An example of this would be reserving a facility which had not been declared. A future enhancement would be to develop a CSIM compiler.

The performance of a CSIM program is similar to that of a C program, except for the process management overhead. Because the program and the run-time support package are all compiled, CSIM programs should execute at acceptable levels of performance. However, it is possible to devise models which will require a great deal of CPU time.

The choice of a simulation language to use in a simulation project is usually based on factors such as availability and familiarity, and not on the suitability of the language to the proposed model. CSIM is useful because it offers an viable alternative to other simulation languages, and attractive because it is process-oriented and because it is a complete programming language. There are situations where these features will make CSIM a sensible choice.

```
Mon Aug  4 16:28:23   CSIM Simulation Report Version 93

Model: CSIM      Time:            0.000
                 Interval:        0.000
                 CPU Time:        2.400 (seconds)

                      Table 1

                 Table Name: expntl

          mean          1.923        min          0.002
          variance      3.619        max         13.359

                 Number of entries 1000

                      Histogram

     Low      -      High       Count   Fraction   Cumulative
              <     0.000 =         0      0.000        0.000
          0.000     0.800 =       344      0.344        0.344
          0.800     1.600 =       218      0.218        0.562
          1.600     2.400 =       157      0.157        0.719
          2.400     3.200 =        91      0.091        0.810
          3.200     4.000 =        62      0.062        0.872
          4.000     4.800 =        43      0.043        0.915
          4.800     5.600 =        23      0.023        0.938
          5.600     6.400 =        19      0.019        0.957
          6.400     7.200 =        17      0.017        0.974
          7.200     8.000 =        14      0.014        0.988
          8.000         <  =        12      0.012        1.000

                 Total  =  1000
```

Figure 10: Example #5 - Sample Output

REFERENCES

Dahl, O.J. and K. Nygaard (1967), *Simula: A Language for Programming and Description of Discrete Event Systems,* Fifth Edition, Norwegian Computing Center, Oslo.

Franta, W.R. (1977), *The Process View of Simulation,* North Holland, New York.

Gordon, G. (1978), *System Simulation (2nd Edition),* Prentice-Hall, Englewood Cliffs, NJ.

Kernighan, B.W. and D.M. Ritchie (1978), *The C Programming Language,* Prentice-Hall, Englewood Cliffs, NJ.

Kiviat, P.J., Vilanueva, R. and H.M. Markovitz (1975), *Simscript II.5 Programming Language,* CACI Inc., Los Angeles.

Law, A.M. and W.D. Kelton (1982), *Simulation Modeling and Analysis,* McGraw-Hill, New York.

MacDougall, M.H. and J.S. McAlpine (1973), Computer System Simulation with ASPOL, *Symposium on the Simulation of Computer Systems,* ACM/SIGSIM, pp. 93-103.

MacDougall, M.H. (1974), Simulating the NASA Mass Data Storage Facility, *Symposium of the Simulation of Computer Systems,* ACM/SIGSIM, pp. 33-43.

MacDougall, M.H. (1975), Process and Event Control in ASPOL, *Symposium of the Simulation of Computer Systems,* ACM/SIGSIM, pp. 39-51.

MacDougall, M.H. (1976), System Level Simulation, *Digital System Design Automation: Languages, Simulation and Data Base,* (M.A. Breuer, Editor), Computer Science Press, Inc., Rockville, MD, pp. 1-115.

Nance, R.E. (1981), The Time and State Relationships in Simulation Modeling, *Communications of the ACM,* **24,** pp. 173-179.

Pritsker, A.A.B. (1974), *The GASP-IV Simulation Language,* John Wiley and Sons, Inc., New York.

Pritsker, A.A.B. and C.D. Pegden (1979), *Introduction to Simulation and SLAM,* Halstead Press, New York.

Saydam, T. (1985), Process-Oriented Simulation Languages, *Simuletter,* **16,** ACM/SIGSIM, pp. 8-13.

AUTHOR'S BIOGRAPHY

HERB SCHWETMAN is a member of the technical staff of the Microelectronics and Computer Technology Corporation (MCC), joining the staff in 1984. Prior to that, he was a professor in the Department of Computer Sciences at Purdue University. He received his Ph.D. in computer sciences from The University of Texas at Austin in 1970. He is past chairman of the ACM Special Interest Group for Measurement and Evaluation (SIGMETRICS).