Proceedings of the 1988 Winter Simulation Conference
M. Abrams, P. Haigh, and J. Comfort (eds.)

Using CSIM to model complex systems

Herb Schwetman
Microelectronics and Computer Technology Corporation
3500 West Balcones Center Drive
Austin, TX 78759

ABSTRACT

Simulation models of complex systems, defined here to be systems with large number of activities or resources or with activities which exhibit complex behavior patterns, make severe requirements on simulation packages. These requirements are in the form of expressiveness (can you express the complexity in the language of the model?) and in the form of accommodation (can the runtime environment handle the large number of simulation objects?). This paper describes CSIM, a process-oriented simulation language based on C, and the features of CSIM which make it useful in developing and executing models of complex systems.

INTRODUCTION

Complex systems, in the context of this paper, are systems which exhibit one or both of two properties: (1) the system is large, with possibly thousands of resources and/or activities, or (2) the behavior of activities is complicated, in the sense that travel routes are complicated or the use of resources is complicated. The ability of a simulation package to model such systems is constrained by a number of factors, including the capacity to accommodate large system models, the ability to execute models in a "reasonable" length of time, and, most importantly, the capability for expressing complicated activity behaviors.

Process-oriented simulation (Franta 1977, Saydam 1985) is a form of discrete event simulation which lets the model implementer focus on the activities being processed by a system, as opposed to the systems events which are the focus of event oriented simulation. Several languages have been designed to support process-oriented simulation, including GPSS (Gordon 1978), Simula (Dahl and Nygaard 1967), SLAM II (Pritsker and Pegden 1979), Simscript II.5 (Law and Kelton, 1982), ASPOL (MacDougall and McAlpine 1973, MacDougall 1974, and MacDougall 1975) RESQ (Sauer and MacNair 1983) and PAWS (IRA 1987). More recently, another process-oriented simulation language, called CSIM, has been introduced (Schwetman 1986).

Whenever a new simulation language or package appears, several questions naturally arise, questions such as:

1. How is this language similar to and, also, different from other such languages? and
2. For what kinds of system models is this language appropriate? or inappropriate?

It is not a goal of this paper to make comparisons between CSIM and other simulation languages. CSIM is different from many other languages in that (1) it is based on the C programming language and (2) it supports the process-oriented simulation paradigm at a fairly high level. Models of many simple systems can be expressed economically (in terms of the number of statements required). However, because CSIM includes all of C, there are few restrictions placed on the size and complexity of the systems which can be efficiently modeled. The net result is that simulation analysts who know the C language and some of the fundamental concepts of discrete event simulation are able to rapidly construct and use simulation models.

This paper gives a very short introduction to CSIM and then explores some of the features which support implementation of models of complex systems. Two examples illustrate some of these features.

CSIM

CSIM extends the capabilities of the C programming language, to allow programmers to write process-oriented simulation programs. While most of the important features of CSIM are described in (Schwetman 1986), a very brief summary of some of these follows.

The basic entity in CSIM a *process* which is a procedure (a C procedure) which executes a *create* statement. Processes are used to model the active components of a system. *Facilities* and *storages* are used to model the passive components of a system. Active processes *reserve* and *release* (or *use*) a facility and *allocate* and *deallocate* portions of a storage. In addition, *events* and *mailboxes* can be used to implement synchronization and communication between active processes. The *hold* statement causes a specified interval of (simulated) time to pass for the process executing the statement. It should be noted that time in a CSIM model time passes only during the execution of *hold* statements by processes.

CSIM has a number of functions for generating samples from some common probability distributions. A set of standard statistics is collected for each facility and storage. In addition, tables, queue-length tables, and histograms can be used to collect and present statistics under program control. A *trace* option can be invoked, to generate a detailed listing of the activities of each process.

CSIM programs are compiled by standard C compilers and execute on VAX computers (either UNIX or VMS), SUN workstations, Sequent Balance systems, and CRAY X/MP computers. A User's Manual (Schwetman 1987) describes all of the

features in more detail. Copies of CSIM can be obtained from the author.

IMPLEMENTING MODELS OF COMPLEX SYSTEMS

The are two major issues which must confronted during the development of a model of a complex system; these are:

1. Can the programmer conveniently and easily express the features of the model in the language? and

2. Does the development environment help the programmer implement and manage a complex program?

The second issue can be dealt with in a straightforward manner: CSIM programs are really C programs, with the addition of the simulation statements. Thus, all of the tools on a system (e.g. a UNIX-based system - UNIX is a trademark of AT&T Bell Labs) which aid the development of C programs are available to the CSIM programmer. Another impact of the availability of C is that math and statistics libraries, as well as libraries of graphics programs, are accessible from CSIM programs. This means that the CSIM programmer can develop a simulation model to meet almost any set of requirements.

The first issue (the expressiveness issue) is a more subjective one. The following observations cast some light on this issue:

- the process-oriented view can, in some cases, simplify the expression of a simulation model; models of many kinds of systems are naturally stated in terms of interacting processes competing for resources;

- CSIM has many features to aid the development of process-oriented models; some of these will be described in the next section;

- the built-in distribution functions, automatic data gathering, and event trace also simplify the development of CSIM programs; and

- because CSIM is a superset of C, there are almost no restrictions on the kinds of algorithms which can be part of a CSIM model; in addition, because CSIM is compiled (as opposed to being interpreted), complex algorithms will execute with the best possible performance.

In addition to the "usual" statements needed for expressing a process-oriented model, CSIM has some additional statements which have been shown to be useful in developing models of complex systems. Some of these enhancements are described in the following subsections.

Facility Sets and Multiserver Facilities

The CSIM statements for declaring, initializing, reserving and releasing facilities are shown in the first example (below). In addition, indexed sets of facilities can also be handled (as in the first example). Another kind of facility is one which has multiple servers but only one queue. In this case, waiting processes are assigned to the first available server. The *facility_ms* statement is used in place of the *facility* or *facility_set* statement, to initialize a multi-server facility. This type of facility is shown in the second example below.

Scheduling Disciplines

Some simulation models may require different scheduling disciplines at different resources in a system model. In this context, a scheduling discipline is the rule which governs allocation of a resource to the several processes which may be requesting access. All processes waiting for a resource are held in a queue at that resource; the entries in this queue are in order of descending priority, with equal priorities going later (in order of arrival). Since the default process priority is 1, this means that the default scheduling discipline is FCFS. Other disciplines can be modeled by adjusting the priority of each requesting process. These other disciplines could include shortest or longest process first (priority based on the service time requirement) and smallest or largest process first (priority based on the "size" of the process).

Other allocation strategies can be found in real systems. One example is preempt-resume scheduling. With this strategy, the priority of an arriving process is compared to the priority of the process holding the facility; if the new process has a higher priority, the currently active process is preempted from the facility and returned to the queue, and the new process is given access to the facility. In CSIM, the remaining amount of the original service time for the preempted process is saved and restored when the preempted process resumes use of the facility. Implementation of preempt-resume allocation (and other strategies mentioned below) uncovered a "messy" detail. It is possible for a process to reserve one facility and then reserve another; if the first facility is allocated using a preempt-resume policy, it is likely that a preempted process is not doing a hold statement, but is in fact on a queue for another facility, or possibly waiting for an event to occur. The point is that preempt-resume as described above assumes that the preempted process is doing a hold operation which may not be the case. To get around this problem, the *use* statement was introduced. At a facility with a nonstandard allocation strategy, a process must *use* the facility for a specified period of time. By limiting access of nonstandard facilities to *use* statements, it can be guaranteed that each process at this facility is locatable and under direct control of the allocation strategy.

In CSIM, the *set_servicefunc* procedure is invoked to establish a nonstandard allocation strategy at a facility. The strategies which are currently implemented are:

a. preempt-resume (pre_res),

b. infinite number of servers (inf_srv),

c. last-come, first-served preempt (lcfs_pr),

d. round robin with time slice (rnd_rob), and

e. processor sharing (prc_shr).

Processor sharing (Baskett et. al. 1975) is a special case of a load dependent service function. With processor sharing, at a

facility with n customers present each customer receives 1/n-th of the capacity of the facility. In CSIM, the load dependent service function used by the processor sharing strategy can be overlaid with other load dependent service functions.

Dynamic Processes and Complex Behavior

CSIM processes have no imposed interrelationships; no process is assumed to be the parent or child of another process. This means that the programmer is able to adopt the process structure which best suits the system being modeled. In many cases, a flat structure, with no direct relationships is the best. Because CSIM processes are created dynamically and can also terminate dynamically, the number and kind of processes can vary in unpredicted ways. Processes can use local events and mailboxes to achieve synchronization between dynamically created processes.

Time-outs (while waiting for messages to arrive or events to occur) represent another form of complex behavior which may be needed in a simulation model. For example, in a model of a communications protocol, a retransmission could be required if an acknowledgment is not received within a specified interval. In CSIM, alternative forms of the *receive*, *wait* and *queue* statements were developed to handle time-outs. These forms (*timed_receive*, *timed_wait* and *timed_queue* respectively) require an additional interval parameter and return a flag value. This flag indicates whether the event occurred (equivalently the message was received) or the interval expired.

EXECUTING MODELS OF COMPLEX SYSTEMS

CSIM has a number of features to facilitate the execution of models of large systems. These models typically have either a large number of facilities and/or storages as well as a large number of processes. All of the data structures associated with facilities, storages and processes are dynamically managed. There are *maximums* which limit the number of facilities, storages and processes which can be simultaneously instantiated; however, these maximums can be changed by CSIM functions, so, in fact, any number of these (up to the limits imposed by the memory on the host system) can be accommodated. Furthermore, the creation of these data structures is done with the goal of good performance. When one of these structures is initialized, a queue of "old" (destroyed) structures is checked first; if an "old" structure is available, it is reused, with a significant saving in execution time. If the queue of "old" structures is empty, the *malloc* function (the memory allocator for C programs) is called, to obtain enough memory for several structures; one is returned for the structure being initialized, and the others are put on the "old" structure queue. Experiments have shown that this strategy can significantly improve performance over other strategies which were tried. All structures for facilities, storages, and processes, as well as for mailboxes, messages and events are managed in this manner.

In a CSIM program, all queues of processes are implemented as doubly linked lists, with both head and tail pointers. In many cases, additions to a queue are always at the tail of the queue and can be done in constant time. The list is searched only when a process does not go at the head or tail of the list. More complex strategies have been investigated, but, so far, have not be put into use.

EXAMPLES

As an example of a large system, consider a transaction processing computer system as shown in Figure 1. In this system, there are 5500 terminals connected to a single computer system. The computer system includes one CPU and 92 disk drives. Each transaction experiences a "think" period, drawn from a negative exponential distribution with mean 30.0 seconds and then enters the computer. At the computer, a transaction cycles through the CPU and randomly selected disk drives, ending with a CPU visit. On the average, a transaction makes 11 visits to the CPU and 10 visits to the collection of disk drives. The listing of the CSIM program which models this system is shown in Figure 3. The outputs from this are shown in Figure 4, and a summary of the computational resources required for a run of this program is given in Table 1. The size of the executable module is estimated to be about 85,000 bytes; using the data in Table 1, it can be estimated that the program required a little less than two megabytes of memory while it was executing.

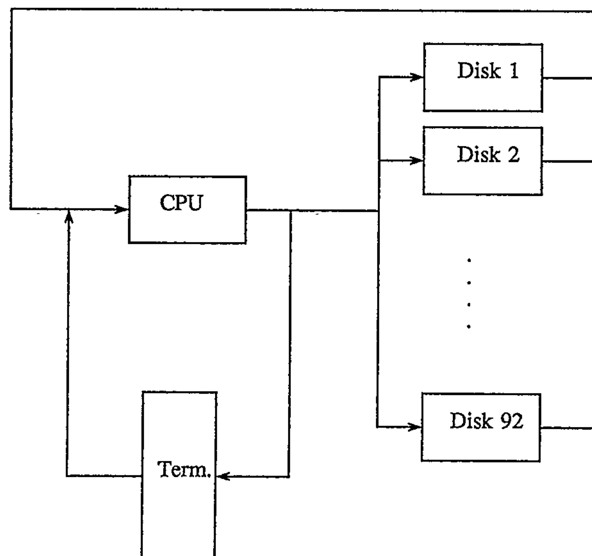

Figure 1: Diagram of First Example

The next example is based on an example which has been often used to illustrate discrete event simulation (Russell 1985). The example models an "African" port which is used to load crude oil into tankers. As shown in Figure 2, the port has three docks. A tugboat is required to position a tanker at a dock for loading and to remove a tanker from a dock after loading. In this example, there are two tugboats. Furthermore, if a tugboat breaks down, another tugboat must tow the broken one to a repair area, where it is repaired and then resumes operation.

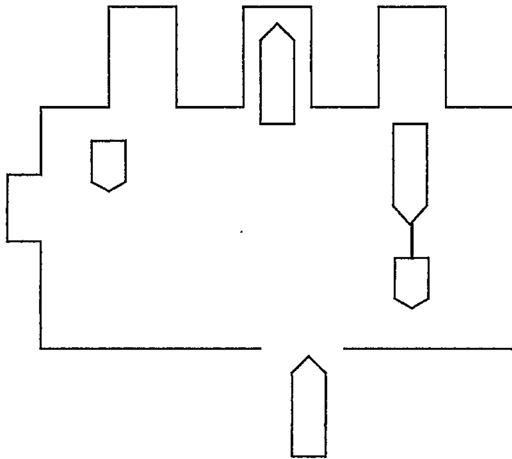

Figure 2: Diagram of Second Example
(The African Port)

The listing of the CSIM program which implements this model is in Figure 5, and the output from a sample run is in Figure 6. This model shows how processes can be used to model active entities. In this program, each tanker is a process. When a tugboat is needed, a *use_tug* procedure is called. This procedure normally just *reserves* a tugboat, does a *hold* for the required amount of time, and then *releases* the tugboat. If the tugboat experiences a breakdown, the function makes a recursive call, to get another (really the other) tugboat, to tow it to the repair facility. Then the new tugboat tows the abandoned tanker to its destination. The repair is modeled as a separate process; when completed, the repaired tugboat resumes offering service to incoming and outgoing tankers.

In this example, there is a potential deadlock. If the second tugboat experiences a breakdown while towing the first (broken) tugboat, then it will try to reserve a tugboat and all of the tugboats will become unavailable. In the model as shown in Figure 5, tankers will continue to arrive, but no further movement of tankers will occur. The program could be modified, to detect this condition.

This example also illustrates another feature of CSIM. Normally, when a process releases a facility, the identity of the process doing the release must be the same as the process which reserved the facility. In the second example, the tanker process (really the use_tug procedure in the tanker process) reserves a tugboat. In the normal case, this process will subsequently release that tugboat. However, in case of a breakdown, the repair process will release that tugboat. The *release_server* statement is used in this situation; this statement is given a server index (returned by the *reserve* statement) and uses it to locate the correct server to release.

SUMMARY AND CONCLUSIONS

This paper has presented some of the features found in the CSIM simulation package which enable CSIM programs to model the structure and behavior of complex systems. In particular, large systems, with large numbers of resources and/or large numbers of processes can be accommodated. In addition, models with dynamic and possibly complex behaviors can be developed. The ability to model dynamic behaviors comes from the flexible process structure. The ability to model complex behavior comes from having the full power of the C programming language as a subset of CSIM.

CSIM has been used to model a wide variety of systems. At MCC, most of these have been models of computer systems. Some of these are described in papers given in the List of References (Alexander et. al. 1986, Alexander and Copeland 1988, and Boughter et. al. 1987). Other CSIM programs have modeled communications protocols, disk subsystems and database systems. In most of these projects, the ability to write C programs which could easily accommodate process oriented simulation turned out to be a powerful tool. The flexibility of having a full featured programming language combined with the ease-of-use of process-oriented simulation has both simplified the development effort and enhanced the usefulness of these models.

REFERENCES

Alexander, W., Keller, T. and E. Boughter, "A Workload Characterization Pipeline for Models of Parallel Systems", *MCC Technical Report,* DB-306-86.

Alexander, W. and G. Copeland, "Comparison of Dataflow Control Techniques in Distributed Data-Intensive Systems", *Proceedings of SIGMETRICS Conference on Computer Performance Modeling, Measurement and Evaluation,* ACM/SIGMETRICS, May, 1988.

Baskett, F., Chandy, M., Muntz, R. and F. Palacios, "Open, Closed and Mixed Networks of Queues with Different Classes of Customers", *Journal of the ACM,* (22), April, 1975, pp. 248-260.

Boughter, E., Alexander, W. and T. Keller, "A Tool for Performance-Driven Design of Parallel Systems", *MCC Technical Report,* ACA-ST-312-87.

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

Franta, W.T., *The Process View of Simulation,* North Holland, New York, 1977.

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

Information Research Associates (IRA), *Paws 3.0 Performance Analyst's Workbench System,* Information Research Associates, 1987.

Law, A. and W. Kelton, *Simulation Modeling and Analysis,* McGraw-Hill, 1982.

MacDougall, M.H. and J.S. McAlpine, "Computer System Simulation with ASPOL", *Symposium on the Simulation of Computer Systems,* ACM/SIGSIM, 1973, pp. 93 - 100.

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

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

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

Russell, E.C., "Simscript II.5 Tutorial", *Proceedings of the 1985 Winter Simulation Conference*, 1985, pp. 57 - 58.

Sauer, C.H. and E.A. MacNair, *Simulation of Computer Communication Systems*, Prentice-Hall, 1983.

Schwetman, H.D., "CSIM: A C-Based, Process-Oriented Simulation Language", *Proceedings of the 1986 Winter Simulation Conference,* 1986, pp. 387 - 396.

Schwetman, H.D., "CSIM Reference Manual", *MCC Technical Report,* ACA-ST-252-87, September, 1987.

Saydam, T., "Process-Oriented Simulation Languages", *Simuletter,* (16 2), ACM/SIGSIM, April 1985, pp. 8 - 13.

AUTHOR'S BIOGRAPHY

HERB SCHWETMAN is a Senior Member of Technical Staff at Microelectronics and Computer Technology Corporation (MCC). He is also an Adjunct Professor Computer Science at The University of Texas at Austin. Prior to joining MCC in 1984, he was on the faculty of Department of Computer Sciences at Purdue University. He received his Ph.D. in Computer Sciences from The University of Texas in 1970. He was Chairman of the ACM Special Interest Group on Measurement and Modeling (SIGMETRICS) from 1981 until 1985.

Herb Schwetman
MCC
3500 West Balcones Center Drive
Austin, Texas 78759
(512) 338-3428


```
/***********************************************/
/*
/* closed model of a computer system with many processes
/*
/***********************************************/

#include "/usr/deathstar/sa/hds/csim/lib/csim.h"

#define NTERM   5500        /* num of terminals */
#define ND      92          /* num of disk drives */
#define P_EXIT  (1.0/11.0)  /* prob of exit to term */
#define S_CPU   0.0005      /* mn time per CPU req */
#define S_DISK  0.030       /* mn DISK service time */
#define Z       30.0        /* mean think time */

#define RUNTIME   200.0     /* simulated running tm */

FACILITY disk[ND], cpu;
TABLE   rt;
EVENT   done;

sim()                           /* sim (main) process */
{
        int i;

        create("sim");
        init();
        for(i = 0; i < NTERM; i++) {/* generate arrivals */
                trans(i);
                }
        wait(done);
        report();
        mdlstat();
}

init()                          /* initialization proced */
{
        max_processes(NTERM+1);
        facility_set(disk, "d", ND);
        cpu = facility("cpu");
        rt = table("resp tm");
        done = event("done");
}

trans(nt)                       /* transaction process */
int nt;
{
        float t, x;
        int i;

        create("trans");
        while(clock < RUNTIME) {
                hold(expntl(Z));
                t = clock;
                do {
                        reserve(cpu);
                                hold(expntl(S_CPU));
                        release(cpu);
                        x = prob();
                        if(x > P_EXIT) {
                                i = random(0, ND-1);
                                reserve(disk[i]);
                                        hold(expntl(S_DISK));
                                release(disk[i]);
                                }
                        } while(x > P_EXIT);
                record(clock - t, rt);
                }
        set(done);
}
```

Figure 3 : Listing of First Example

Wed Apr 27 17:15:23 1988 CSIM Simulation Report Version 121

Model: CSIM

Time:	200.001
Interval:	200.001
CPU Time:	1593.033 (seconds)

Facility Usage Statistics

			means					counts	
facility	srv	disp	serv_tm	util	tput	qlen	resp	cmp	pre
d[*]			0.030	52.987	1766.4	124.246	0.070	353233	0
cpu			0.000	0.968	1943.9	23.909	0.012	388782	0

Table 1
Table Name: resp tm

mean	0.831	min	0.000
variance	0.800	max	9.334

Number of entries 35434

Figure 4: Output from First Example

CSIM Model Statistics

CPU Time:	1593.450 (seconds)
Memory (malloc):	1904012 (bytes)
Processes	
Started:	5501
Saved:	1167644
Terminated:	1
High water mark:	5501
Stacks	
Allocated:	5501
High water mark:	341060 (words)
Average:	61 (words)
Maximum:	62 (words)
Current:	340998 (words)

Table 1: Resource Usage by First Example

```
/**********************************************/
/*
/* CSIM model of African port with tugboats/*
*/
/**********************************************/

#include "/usr/deathstar/sa/hds/csim13/lib/csim.h"

#define SIMTIME     365*24.0    /* length of run */
#define NDOCKS      3           /* num of docks */
#define NTUGS       2           /* num of tugboats */
#define NCLASS      3           /* num of classes of tnkrs */
#define IARTM       11.0        /* mn tnkr interarrival tm */
#define TUG_HAUL 1.0            /* mean tug haul tm */

#define PROB_BRKDWN     0.01 /* prob of tug breakdown */
#define REPAIR_TM       3.0  /* mean repair time */
#define TUG_TOW         0.5  /* mn tm to tow tug */

FACILITY dock;
FACILITY tugboat;
QHIST   repairs;
float pc[] = {0.25, 0.55, 0.20};    /* tnkr class probabilities */
float st[] = {18.0, 24.0, 36.0};    /* mn load tms per class */

sim()                               /* sim (main) process */
{
        create("sim");
        init();
        while(clock < SIMTIME) { /* generate arrivals */
                tanker(select());
                hold(expntl(IARTM));
                }
        report();
}

tanker(c)                           /* tanker process */
int c;
{
        create("tanker");
        reserve(dock);
                use_tug(TUG_HAUL);
                hold(expntl(st[c]));
        release(dock);
        use_tug(TUG_HAUL);
}

use_tug(t)                          /* use tugboat procedure */
float t;
{
        int tugi;
        tugi = reserve(tugboat);
        if(prob() < PROB_BRKDWN) {
                use_tug(TUG_TOW);
                repair(tugi);
                use_tug(t);
                return;
                }
        hold(expntl(t));
        release_server(tugboat, tugi);
}

repair(tugi)                        /* repair-tugboat process */
int tugi;
{
        create("repair");
        note_entry(repairs);
        hold(expntl(REPAIR_TM));
        release_server(tugboat, tugi);
        note_exit(repairs);
}

init()                              /* initialization proced */
{
        dock = facility_ms("dock", NDOCKS);
        tugboat = facility_ms("tug", NTUGS);
        repairs = qhistogram("repair fac.", NTUGS+1);
}

int select()                        /* rand class-select func */
{
        int i;  float x, y;
        x = prob();
        i = 0;
        y = pc[i];
        while(x > y) {
                i++;
                y += pc[i];
                }
        return(i);
}
```

Figure 5: Listing of Second Example

Fri Jul 29 09:34:17 1988 CSIM Simulation Report Version 131

Model: CSIM

Time: 8815.095
Interval: 8815.095
CPU Time: 2.967 (seconds)

Facility Usage Statistics

facility	srv	disp	serv_tm	util	tput	qlen	resp	cmp	pre
			means					counts	
dock	1		24.505	0.848	0.0			305	
dock	2		25.323	0.787	0.0			274	
dock	3		29.249	0.737	0.0			222	
dock			26.099	2.372	0.1	6.272	69.027	801	0
tug	1		1.052	0.121	0.1			1017	
tug	2		1.058	0.075	0.1			623	
tug			1.054	0.196	0.2	0.206	1.107	1640	0

QTable 1

QTable Name: repair fac.

Mean queue length	0.010	Max queue length	1
Mean time in queue	4.628	Number of entries	19

Queue Table Histogram

Length	% of Elapsed Time	Cumulative	Count	Mean Time
0	0.990	0.990	19	459.324
1	0.010	1.000	19	4.628

Figure 6: Output from Second Example