SIMPACK: GETTING STARTED WITH SIMULATION PROGRAMMING IN C AND C++

Paul A. Fishwick

Dept. of Computer & Information Science
University of Florida
Bldg. CSE, Room 301
Gainesville, FL 32611

ABSTRACT

SimPack is a collection of C and C++ libraries and executable programs for computer simulation. In this collection, several different simulation algorithms are supported including discrete event simulation, continuous simulation and combined (multimodel) simulation. The purpose of the *SimPack* toolkit is to provide the user with a set of utilities that illustrate the basics of building a working simulation from a model description. We demonstrate that special purpose simulation programming languages can be easily constructed using language translation software with the SimPack utilities which act as the "assembly language." We present several different dynamical system model forms and overview the methods used in *SimPack* for executing these models. *SimPack* includes some fairly extensive simulation facilities, and is in use by various instructors, researchers and industrial analysts for their modeling and simulation experiments.

1 INTRODUCTION

Computer simulation is highly interdisciplinary and simulation users and researchers can be found, for instance, in physics, industrial engineering, operations research and computer science. Regardless of this diversity and apparent global coverage, there are definite generic aspects of simulation that form the core of the simulation discipline. The core is composed primarily of three areas: 1) model design, 2) model execution and 3) input and output analysis. We have constructed a toolkit called *SimPack*[1] that permits experiments in the first area: model design. *SimPack* is a set of C and C++ tools supporting the creation of executable models. Before discussing *SimPack* in specific, we briefly discuss the background of software support available for simulation.

Most simulation packages cover one of two areas: discrete event or continuous. Discrete event methods cater to those performing modeling of queuing networks, flexible manufacturing systems and inventory practices. Continuous methods are normally associated with block diagram methods for control and system engineering. Some available software can perform both types of simulation; however, bulk support is usually available in only one form. Classic discrete event simulation languages such as GPSS (Schriber 1991), SLAM (Pritsker 1986), SIMSCRIPT (Markowitz, Kiviat and Villaneuva 1987) and SIMAN (Pegden, Sadowski and Shannon 1990) have been used extensively over the past decade, while continuous languages such as CSMP, DESIRE (Korn 1989) and continuous system language derivatives provide adequate support for modeling continuous systems (Cellier 1991) in the form of block models. In addition to simulation languages, there are toolkits or libraries such as CSIM (Schwetman 1988; Schwetman 1990), SMPL (MacDougall 1987) and SIM++ (Lomow and Baezner 1990) which are C and C++ based libraries for discrete event simulation. SMPL is based in C while CSIM and SIM++ contain class libraries for programming in C or C++.

Given the plethora of simulation languages and toolkits, we want to address our reasons for creating *SimPack*:

1. *To increase the variety of available model types* — Few packages permit the variety of model types available in *SimPack*. As systems become large and complex, the analyst will require simulation software that can support a wide variety of model types. One solution to modeling complex systems in simulation languages is to convert all models into that language. Our approach is quite different — we recognize that, for instance, resource contention is best modeled with Petri nets; queuing problems are best mod-

[1] To obtain a copy of the source code, note the instructions given in appendix A.

eled with queuing graphs and some continuous systems are best modeled with engineering block diagrams. Therefore, our approach is to provide a set of C and C++ tools that accommodates a direct translation from these unique graphing approaches into callable routines; we do not force the user to think in terms of a single overall language for all simulation applications. Instead, we believe that most systems will contain model components whose types are quite different. The perceived need to have an "all in one" simulation language does not match most real world problems where a set of well-tested model types has developed naturally.

2. *To create template algorithms for many cases* — We found that —for students and professionals alike— it is sometimes difficult to know exactly where to start when creating a simulation for a particular application. For instance, how does one efficiently simulate a system of difference equations, or how does one combine discrete event and continuous models? What is needed is a "handle" or set of template algorithms so that the analyst can see how to solve a simple problem of a specific type. This "seed software" approach eases the inertial problems associated with determining where to begin analysis.

3. *To avoid learning a special language syntax* — Researchers comfortable with a popular language such as C or C++ can immediately begin creating simulations without having to learn an entirely new language syntax. In this sense, we were inspired by both CSIM and SMPL[2].

4. *To illustrate the relationship between event and process oriented discrete event simulation* — Even though some C based simulation libraries cater either to "process-oriented" or "event-oriented" simulation, we prefer to think of *process* orientation as being one level higher than event orientation in the sense that those interested in programming at the process level can use a *SimPack* tool such as *miniGPSS* which compiles process code into event code since this must be done in any circumstance. In other words, the event orientation provides the "assembly language" in *SimPack* and process orientation is achieved by building software that hides this level using text translation.

5. *To develop a freeware simulation package* — There are few nominal cost or freeware simulation packages available to researchers or industry analysts. Additionally, some packages are extremely expensive, making the entire *approach* of simulation untenable for the average analyst. Some special student simulation packages are available; however, these are intentionally limited by speed or size of model.

[2]The SMPL syntax for functions was used as a basis for the queuing library within SimPack.

Our approach was to build a freeware simulation toolkit that would support simulation development for a wide variety of modeling types including the following:

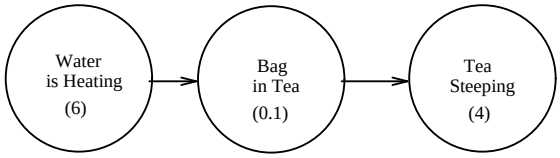
- Declarative Models: An emphasis on explicit state to state changes as found in finite state automata (FSA) and Markov models.
- Functional Models: A focus on "function" or "procedure" as in queuing networks, block models, pulse processes and stochastic Petri nets.
- Constraint Models: Defined by differential and difference equations.
- Multimodels: Defined as conglomerates of other models connected in a graph or network to solve combined simulation problems at multiple abstraction levels (Fishwick 1991; Fishwick and Zeigler 1992).

The goal of *SimPack* is not to create highly polished programs with refined interfaces (although the program XSimCode has a user friendly interface for X windows). Instead, analysts can start with the tool that best represents the solution to the modeling problem at hand and then build upon that "seed program" to create a comprehensive simulation language or program tailored to the user's specific needs.

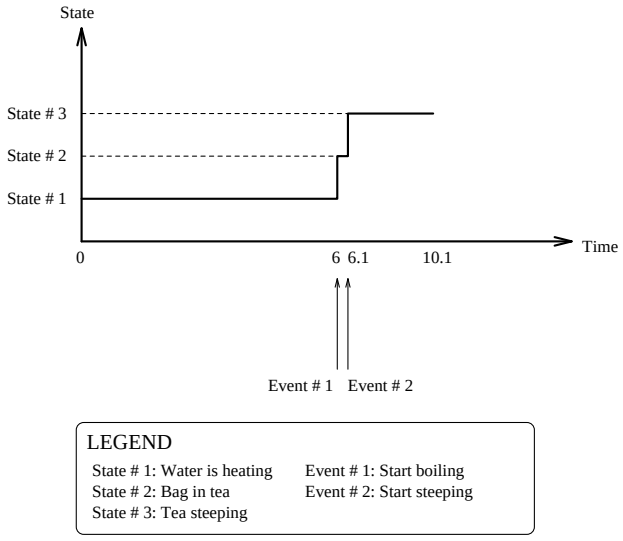
2 DECLARATIVE MODELS

2.1 Overview

Declarative models focus on a change in state; therefore, the concept of *state description* plays a vital role. FSA and Markov models are simple examples of how a declarative model can represent a system at a high level of abstraction. Each model type requires the specification of a finite number of state descriptors. For the FSA, the input guides the state to state transition changes, while for the Markov chain, there is an element of non-determinism where arcs specify the probability that a given transition will be taken. States, in each model type, specify a system state for some interval of time which must be specified. FSAs and Markov models are used to represent a system at a high abstraction level; states are often



(a) Tea System Model



(b) Tea System Trajectory

Figure 1: Tea Model

synonymous with *phases*. Because of the focus on "high level" modeling, these model types are associated with coarse grained control of a lower level system. Consider figure 1(a): this is a model for brewing tea. When we execute this model, we obtain a trajectory shown in fig. 1(b). The model in fig. 1(a) has been created by *coupling* three states together. The *SimPack* program for simulating this system takes, as input, a data file containing the topology of the FSA. The file contains 1) the number of states, 2) the state interval times, 3) state descriptor and the transition state based on an input $i \in \{1, 2, 3, \ldots\}$, and 4) an input signal (i.e., string) to control the FSA:

```
3
6.0 0.1 4.0
0  1
1  2
2  3
3  99
WATER_IS_HEATING
BAG_IN_TEA
TEA_STEEPING
3 111
```

By itself, the FSA may seem like a very simple modeling technique that would rarely be used; however, we stress its importance in the section on multimodeling (sec. 5).

2.2 Implementation Aspects

The implementations of the FSA and Markov model use a transition table. In C, the transition table is stored as a simple two dimensional array, whereas in C++, the transition function is a method for an FSA object that has an attribute of *system state*. In cases where an FSA controls other abstraction levels, the C++ implementation uses a network of objects where each object is an FSA state and the methods associated with the state represent the lower level semantics operating over a state space where the FSA state (phase) serves as a partition.

3 FUNCTIONAL MODELS

3.1 Overview

Whereas declarative models emphasize the change in state, functional models are composed of networks of coupled functions. This represents a fairly large class of models including queuing networks for discrete signals, engineering block models for continuous signals and a small version of GPSS called miniGPSS. The object model for the C++ queuing library within *SimPack* is shown in figure 2. The C queuing library is similar without the capability for object encapsulation. So, for instance, to request a facility in C, we use `request(facility,token,priority)` where a `token` with a certain `priority` requests use of a `facility`. In C++, the request is in the form of a method attached to a facility object: `facility.request(token,priority)`. As an example system consider a CPU/Disk system illustrated in (MacDougall 1987) and shown in figure 3. There is a single CPU (Central Processing Unit) that will serve to execute a set number of tasks. Each task will have disk I/O requirements. The cyclical nature of this model reflects the fact that a single task will be simulated as follows:

```
Use CPU
Access a disk (1-4)
Use CPU
Access a disk (1-4)
Use CPU
```

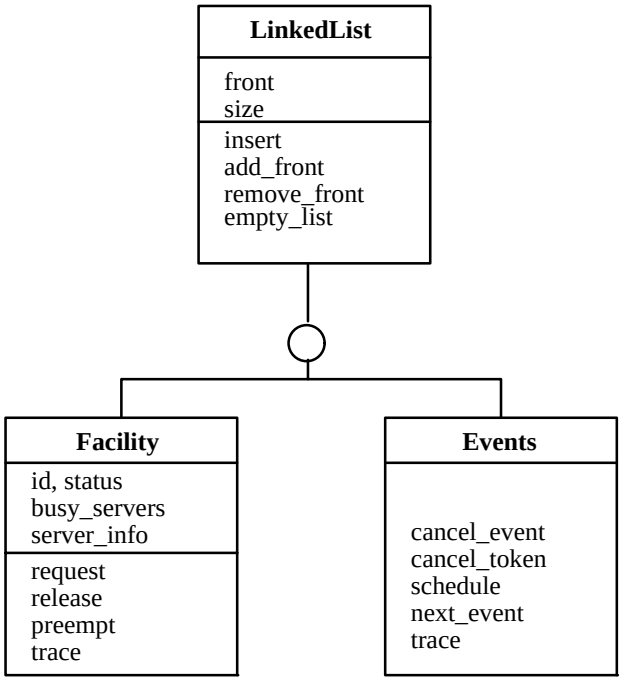


Figure 2: Queuing Object Class Library

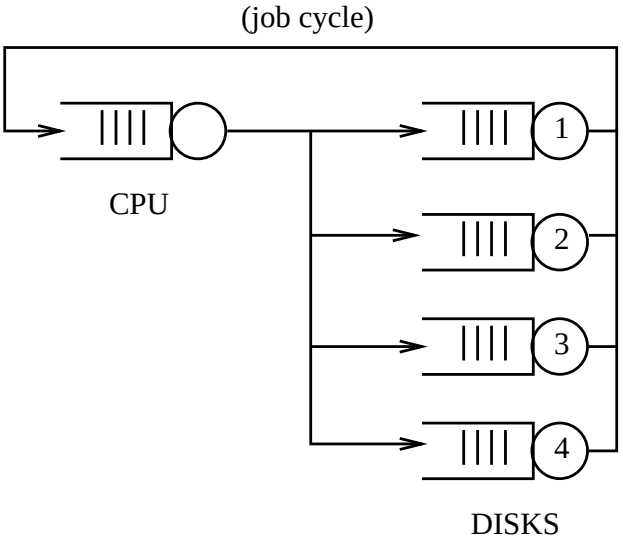


Figure 3: CPU with Four Disk Units

```
Access a disk (1-4)
... etc ...
```

This cyclical pattern is fairly typical of most programs that alternately access the CPU and mass storage. In the simulation, there are a total of nine jobs (each modeled as a token) where six of the jobs are class zero (low priority) and the remaining three jobs are class one (high priority). While the disk units are simply requested by the cycling jobs, the CPU can be preempted if the current CPU job is of a lower priority than the job requesting service. The simulation proceeds until a set number of *tours* have been completed. One tour for a task is the use of the CPU and then the disk before cycling back.

3.2 Implementation Aspects

The CPU/Disk system was first coded in C and then, subsequently, in C++. The C implementation is shown in appendix B and the C++ code is in appendix C. Appendix D displays the statistical output from the simulation.

4 CONSTRAINT MODELS

The constraint part of *SimPack* includes capabilities for modeling 1) difference equation systems, 2) differential equation systems, and 3) delay differential equation systems. In most cases, the most uniform method of simulation is to convert the equation(s) into first order form and then to simulate the system by updating the state vector. For example, assume that we have a single, multiple order ODE with variable x:

$$\frac{d^n x}{dt^n} = F(x, \frac{dx}{dt}, \frac{d^2x}{dt^2}, \ldots, \frac{d^{n-1}x}{dt^{n-1}}).$$

This represents an ODE with an arbitrary order n. Initial conditions for this equation are as follows (where $t_0 = 0.0$ is common):

$$x(t_0), \frac{dx}{dt}(t_0), \frac{d^2x}{dt^2}(t_0), \ldots, \frac{d^{n-1}x}{dt^{n-1}}(t_0).$$

We must convert this equation to a set of **first order** differential equations by creating new variables as follows:

$$x_1(t) = x, x_2(t) = \frac{dx}{dt}, \ldots, x_n(t) = \frac{d^{n-1}x}{dt^{n-1}}.$$

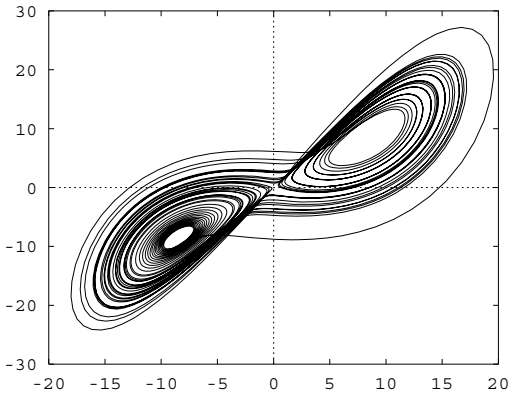


Figure 4: The Lorenz System

Then the generated set of equations is:

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= x_3 \\ &\vdots \\ \dot{x}_{n-1} &= x_n \\ \dot{x}_n &= F(x_1, x_2, \ldots, x_n). \end{aligned}$$

For multiple equations, one creates additional x_i variables and proceeds to break the system down into first order form as shown. The *SimPack* C implementation of the above step includes using two arrays IN and OUT where $\dot{x}_1 = IN[1]$, $\dot{x}_2 = IN[2]$, $\dot{x}_3 = IN[3]$, and so on. The IN array stores all of the derivative values. Likewise, an OUT array stores all the state variable values: $x_1 = OUT[1]$, $x_2 = OUT[2]$, $x_3 = OUT[3]$. Now, we can think of the *integration* step for a variable $\dot{x}_i$ as simply taking the value in cell $IN[i]$, integrating it and placing this new value into cell $OUT[i]$.

Consider the Lorenz system:

$$\begin{aligned} \dot{x}_1 &= \sigma(x_2 - x_1) \\ \dot{x}_2 &= (1 + \lambda - x_3)x_1 - x_2 \\ \dot{x}_3 &= x_1 x_2 - b x_3 \end{aligned}$$

with parameter values of $\sigma = 10$, $\lambda = 24$, and $b = 2$, and initial conditions of $x_i(0) = 1.0$. The resulting projection of (x_2, x_3) shown in figure 4 shows a chaotic attractor. To plot the solution to difference and differential equations, we have used two packages *gnuplot* (from the GNU Free Software Foundation) and *xvgr*.

The implementation procedure for difference equations is almost identical where the input is a signal $X(t)$ and the output is a delayed signal $X(t-1)$. However, for difference equations, an extremely convenient implementation method is to implement a circular queue to "remember" previous values of X. This is efficient, since a pointer is moved rather than values being copied when an update of state variables is needed.

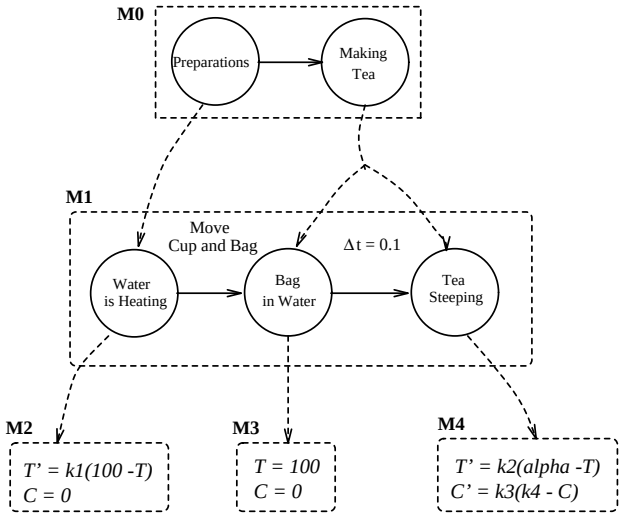


Figure 5: Abstraction Network with Three Levels

To solve both difference and differential equations using the first order (canonical) method, there are three basic steps in the algorithm:

1. Set initial conditions for the state vector.
2. Update state equations (using the given equations).
3. Delay or Integrate all state variables.

5 MULTIMODELS

5.1 Overview

Multimodels are networks of models where each component model is most often a level of abstraction for a system. Consider the tea system presented in the earlier FSA section (sec. 2). FSAs can be used to "control" the phase transitions associated with making tea. For instance, in figure 5, we see a three level multimodel that contains two FSAs and a set of 3 low level differential equation sets whose state space is $C \times T$ (C is the concentration of tea, and T is the temperature).

We assign each phase to a geometrical partition π where $\pi \subset C \times T$. The homomorphic relationship of $M1$ to $M2$, $M3$ and $M4$ is most easily displayed using a "phase graph" as shown in figure 6.

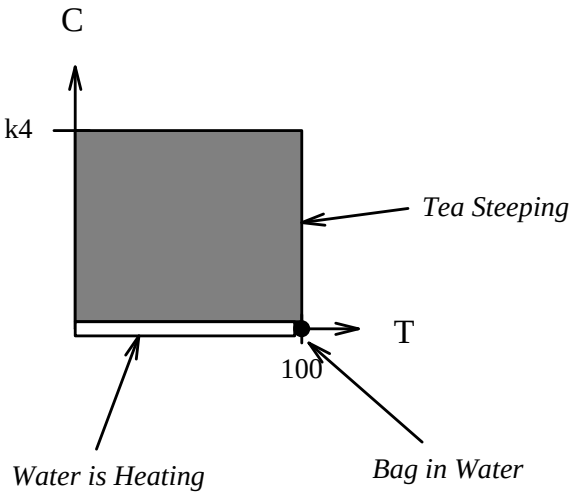


Figure 6: Three Phases of the Tea System

5.2 Implementation Aspects

Multimodels are most naturally expressed using object oriented design since the object model for the tea system is isomorphic to the multimodel as pictured in figure 5. Specifically, note the following:

1. FSA $M0$ contains two objects, one for each state. The inter-state connectivity is encoded using pointers to objects (i.e., a network). Connections to $M1$ are located in the methods of objects in $M0$.
2. FSA $M1$ is encoded in the same way as $M0$. Connections of each state object to $M2$, $M3$ and $M4$ are specified using methods (for solving the equations) and pointers.

6 CONCLUSIONS

We have presented an overview of *SimPack* and its capabilities. Our chief motivations for the construction of *SimPack* stem from the need for analysts to have an easy-to-obtain toolkit that provides them with a starting point for simulating a system. Most real-world simulations will be larger and more complicated than the code provided within *SimPack*; however, the provided code can serve as a template or building block upon which a more comprehensive application can be constructed. *SimPack* has been used in simulation classes at the University of Florida for the past two years with much success. Students, especially within computer science, are already familiar with structured programming languages such as Pascal and C and, therefore, it is most natural for them to construct models and simulations without resorting to learning a new simulation language syntax.

SimPack was originally coded in C and many parts of the code have been ported to C++ since the object oriented approach to design is a natural extension of modeling: models contain components, component information and interactions — which correspond roughly to objects, attributes and methods. We are continuing to investigate new C++ approaches to modeling including the use of the *template* capability for unifying the simulation operation given a multitude of object types. We are also including *SimPack* as part of an upcoming senior/graduate level textbook (Fishwick 1992) which focuses on modeling.

ACKNOWLEDGMENTS

During the course of developing *SimPack*, several students made substantial contributions. Brian Harrington developed *Xsimcode* which is an X window interface that permits the user to create queuing models which are then translated into C calls in the queuing library. David Bloom wrote *miniGPSS* which serves as a compiler for a scaled-down version of GPSS.

Appendix A: Obtaining a Copy of SimPack

Simpack Version 2.0 is available via anonymous ftp from bikini.cis.ufl.edu (cd to pub/simdigest/tools, specify 'binary' and get simpack-2.0.tar.Z). Read the file VERSION2 to see the enhancements over version 1.0.

Appendix B: CPU/Disk Simulation in C

```
/* NOTE: This program is the SimPack equivalent
of an SMPL program by M. H. MacDougall,
"Simulating Computer Systems", MIT Press, 1987 */


#include "../../queuing/queuing.h"

#define n0 6            /* no. class 0 tasks  */
#define n1 3            /* no. class 1 tasks  */
#define nt n0+n1        /* total no. of tasks */
#define nd 4            /* no. of disk units  */
#define qd 1            /* queued req. return */

#define BEGIN_TOUR 1
#define REQUEST_CPU 2
#define RELEASE_CPU 3
#define REQUEST_DISK 4
#define RELEASE_DISK 5

struct token
  {
    int cls;          /* task's class (& priority) */
    int un;           /* disk for current IO req.  */
```

```
    double ts;          /* tour start time stamp      */
  } task[nt+1];

TOKEN a_token;

int
  disk[nd+1],          /* disk facility descriptors */
  cpu,                 /* cpu facility descriptor   */
  nts=500;             /* no. of tours to simulate  */

double
  tc[2]={10.0,5.0},  /* class 0,1 mean cpu times  */
  td=30.0, sd=2.5;   /* disk time mean, std. dev. */

main()
{
 int icount,i,j,event,n[2]; double t,s[2],rn;
 struct token *p;

 n[0]=n[1]=0; s[0]=s[1]=0.0;
 for (i=1; i<=nt; i++) task[i].cls=(i>n0)? 1:0;
 init_simpack(LINKED);
 cpu=create_facility("CPU",1);
 for (i=1; i<=nd; i++)
  disk[i]=create_facility("disk",1);
 for (i=1; i<=nt; i++) {
  a_token.attr[0] = (float) i;
  schedule(BEGIN_TOUR,0.0,a_token);
 } /* end for */
 icount = 0;
 while (nts) {
   icount++;
   next_event(&event,&a_token);
   i = (int) a_token.attr[0];
   p = &task[i];
   switch(event) {
   case BEGIN_TOUR:  /* begin tour */
     a_token.attr[0] = (float) i;
     p->ts=time();
     schedule(REQUEST_CPU,0.0,a_token);
     update_arrivals();
     break;
   case REQUEST_CPU:  /* request cpu */
     j=p->cls;
     a_token.attr[0] = (float) i;
     if (preempt(cpu,a_token,j) == FREE) {
       rn = expntl(tc[j]);
       a_token.attr[0] = (float) i;
       schedule(RELEASE_CPU,(float) rn,a_token);
     }
     break;
   case RELEASE_CPU:  /* release cpu, select disk */
     a_token.attr[0] = (float) i;
     release(cpu,a_token); p->un=random(1,nd);
     schedule(REQUEST_DISK,0.0,a_token);
     break;
   case REQUEST_DISK:  /* request disk */
     a_token.attr[0] = (float) i;
     if (request(disk[p->un],a_token,0) == FREE) {
       rn = erlang(td,sd);
       a_token.attr[0] = (float) i;
       schedule(RELEASE_DISK,(float) rn ,a_token);
     }
     break;
   case RELEASE_DISK:  /* release disk, end tour */
     a_token.attr[0] = (float) i;
     release(disk[p->un],a_token); j=p->cls;
     t=time(); s[j]+=t-p->ts; p->ts=t; n[j]++;
     update_completions();
     a_token.attr[0] = (float) i;
     schedule(BEGIN_TOUR,0.0,a_token); nts--;
   break;
   }
   }
  report_stats(); printf("\n\n");
  printf("class 0 tour time = %.2f\n",s[0]/n[0]);
  printf("class 1 tour time = %.2f\n",s[1]/n[1]);
}
```

Appendix C: CPU/Disk Simulation in C++

```
/* NOTE: This program is the SimPack++ equivalent
of an SMPL program by M. H. MacDougall, "Simulating
Computer Systems", MIT Press, 1987 */

#include "../../queuing/queuing.h"

#define n0 6          // no. class 0 tasks
#define n1 3          // no. class 1 tasks
#define nt n0+n1      // total no. of tasks
#define nd 4          // no. of disk units
#define qd 1          // queued req. return

enum {BEGIN_TOUR = 1,REQUEST_CPU,
      RELEASE_CPU, REQUEST_DISK,
      RELEASE_DISK};

struct token_info
  {
    int cls;          // task's class (& priority)
    int un;           // disk for current IO req.
    double ts;        // tour start time stamp
  } task[nt+1];

Token a_token;
Facility* disk[nd+1];// disk facility pointers

int
  nts=500;            // no. of tours to simulate

double
  tc[2]={10.0,5.0},  // class 0,1 mean cpu times
  td=30.0, sd=2.5;   // disk time mean, std. dev.

main()  {
 int icount,i,j,event,n[2]; double t,s[2],rn;
 struct token_info *p;
```

```
n[0]=n[1]=0; s[0]=s[1]=0.0;
for (i=1; i<=nt; i++) task[i].cls=(i>n0)? 1:0;
init_simpack(LINKED);
Facility cpu(0,1);
for (i=1; i<=nd; i++) disk[i] =
 new Facility(i,1);
for (i=1; i<=nt; i++) {
 a_token.id = i;
 event_list.schedule(BEGIN_TOUR,0.0,a_token);
} /* end for */
icount = 0;
while (nts) {
 icount++;
 event_list.next_event(event,a_token);
 i = a_token.id;
 p = &task[i];
 switch(event) {
 case BEGIN_TOUR:
   a_token.id = i;
   p->ts=time();
   event_list.schedule(REQUEST_CPU,0.0,a_token);
   update_arrivals();
   break;
 case REQUEST_CPU:
   j=p->cls;
   a_token.id = i;
   if (cpu.preempt(a_token,j) == FREE) {
   rn = expntl(tc[j]);
   a_token.id = i;
   event_list.schedule(RELEASE_CPU,rn,a_token);
   }
   break;
 case RELEASE_CPU:
   a_token.id = i;
   cpu.release(a_token); p->un=random(1,nd);
   event_list.schedule(REQUEST_DISK,0.0,a_token);
   break;
 case REQUEST_DISK:
   a_token.id = i;
   if (disk[p->un]->request(a_token,0) == FREE) {
    rn = erlang(td,sd);
    a_token.id = i;
    event_list.schedule(RELEASE_DISK,rn,a_token);
   }
   break;
 case RELEASE_DISK:
   a_token.id = i;
   disk[p->un]->release(a_token); j=p->cls;
   t=time(); s[j]+=t-p->ts; p->ts=t; n[j]++;
   update_completions();
   a_token.id = i;
   event_list.schedule(BEGIN_TOUR,0.0,a_token);
   nts--;
   break;
   }
 }
 report_stats(); printf("\n\n");
 printf("class 0 tour time = %.2f\n",s[0]/n[0]);
  printf("class 1 tour time = %.2f\n",s[1]/n[1]);
}
```

Appendix D: Summary Statistics for CPU/Disk Model

Note that system utilization is defined as an average over all facilities. Facility 0 is the CPU while facilities 1,2,3 and 4 are the four disks. Service for the CPU is based on an exponential distribution with a mean of $\mu = 10.0$ for class 0 jobs and $\mu = 5.0$ for the higher priority class 1 jobs. Service for the disks is based on an Erlang distribution with $\mu = 30.0$ and $\sigma = 2.5$.

```
+---------------------------+
| SimPack SIMULATION REPORT |
+---------------------------+

Total Simulation Time: 4772.292969
Total System Arrivals: 508
Total System Completions: 500

System Wide Statistics
----------------------
System Utilization: 78.8%
Arrival Rate: 0.106448, Throughput: 0.104771
Mean Service Time per Token: 7.525242
Mean # of Tokens in System: 9.000001
Mean Residence Time for each Token: 85.901283

Facility Statistics
-------------------
F 0 : Idle: 18.3%, Util: 81.7%, Preemptions: 127
F 1 : Idle: 20.8%, Util: 79.2%, Preemptions: 0
F 2 : Idle: 12.0%, Util: 88.0%, Preemptions: 0
F 3 : Idle: 29.3%, Util: 70.7%, Preemptions: 0
F 4 : Idle: 25.4%, Util: 74.6%, Preemptions: 0


class 0 tour time = 93.01
class 1 tour time = 72.73
```

REFERENCES

Cellier, F. E. 1991. *Continuous System Modeling.* Springer Verlag.

Fishwick, P. A. 1991. Heterogeneous Decomposition and Coupling for Combined Modeling. In *1991 Winter Simulation Conference*, pages 1199 – 1208, Phoenix, AZ.

Fishwick, P. A. 1992. *Computer Simulation Modeling: Methodology, Algorithms and Programs.* (to be published as a textbook).

Fishwick, P. A. and Zeigler, B. P. 1992. A Multimodel Methodology for Qualitative Model Engineering.

ACM Transactions on Modeling and Computer Simulation, 1(2).

Korn, G. A. 1989. *Interactive Dynamic System Simulation*. McGraw Hill.

Lomow, G. and Baezner, D. 1990. A Tutorial Introduction to Object-Oriented Simulation and Sim++. In *1990 Winter Simulation Conference*, pages 149 – 153, New Orleans, LA.

MacDougall, M. H. 1987. *Simulating Computer Systems: Techniques and Tools*. MIT Press.

Markowitz, H. M., Kiviat, P. J., and Villaneuva, R. 1987. *Simscript II.5 Programming Language*. CACI, Inc., Los Angeles, CA.

Pegden, C. D., Sadowski, R. P., and Shannon, R. E. 1990. *Introduction to Simulation using SIMAN*. Systems Modeling Corporation, Sewickley, PA.

Pritsker, A. A. B. 1986. *Introduction to Simulation and SLAM II*. Halsted Press.

Schriber, T. J. 1991. *An Introduction to Simulation using GPSS/H*. John Wiley.

Schwetman, H. 1988. Using CSIM to Model Complex Systems. In *1988 Winter Simulation Conference*, pages 246 – 253, San Diego, CA.

Schwetman, H. 1990. Introduction to Process-Oriented Simulation and CSIM. In *1990 Winter Simulation Conference*, pages 154 – 157, New Orleans, LA.

AUTHOR BIOGRAPHY

PAUL A. FISHWICK is an associate professor in the Department of Computer and Information Sciences at the University of Florida. He received the BS in Mathematics from the Pennsylvania State University, MS in Applied Science from the College of William and Mary, and PhD in Computer and Information Science from the University of Pennsylvania in 1986. He has experience at Newport News Shipbuilding and Dry Dock Co. (doing CAD/CAM parts definition research) and at NASA Langley Research Center (studying engineering data base models for structural engineering). His research interests are in computer simulation modeling and analysis methods for complex systems. He is a member of IEEE, IEEE Society for Systems, Man and Cybernetics, IEEE Computer Society, The Society for Computer Simulation, ACM and AAAI. Dr. Fishwick was chairman of the IEEE Computer Society technical committee on simulation (TCSIM) for two years (1988-1990) and he is on the editorial boards of several journals including the *ACM Transactions on Modeling and Computer Simulation*, *The Transactions of the Society for Computer Simulation*, *International Journal of Computer Simulation*, *Simulation* and the *Journal of Systems Engineering*.