Proceedings of the 1995 Winter Simulation Conference
ed. C. Alexopoulos, K. Kang, W. R. Lilegdon, and D. Goldsman

OBJECT-ORIENTED SIMULATION MODELING WITH C++/CSIM17

Herb Schwetman

Mesquite Software, Inc.
3925 West Braker Lane
Austin, TX 78759-5321, USA

ABSTRACT

C++/CSIM17 is toolkit for constructing process-oriented, discrete-simulation models by writing C++ programs. The use of C++ as the implementation language for the model means that the majority of the object-oriented techniques and methodologies that have been developed for C++ programmers can be applied to the design and implementation of simulation models.

1 INTRODUCTION

Programmers have embraced object-oriented methodologies and languages to enhance their ability to design, implement and deliver powerful, user-friendly applications to their customers. Those developing simulation models have the same requirements and, thus, need to leverage these new tools and technologies to deliver simulation models to their customers.

C++/CSIM17 is a toolkit for developing process-oriented, discrete-event simulation models using object-oriented development methodologies. It provides a C++ interface to the capabilities and features of the proven CSIM17 toolkit, used by C programmers for many years (Schwetman 1986, 1988, 1990, 1994).

This paper presents C++/CSIM17. It then shows how the object-oriented features of C++/CSIM17 can be used to implement accurate, efficient simulation models of complex systems. The focus is on using object refinement to add important details to selected components of a model. Object inheritance is also illustrated.

2 OBJECT-ORIENTED SIMULATION MODELS

Object-oriented design is an approach to designing software in terms of objects, where an object is an instance of a class. A class is a collection of data items and methods (procedures and functions) which implement a software "component".

A simulation model of a system is a collection of simulated resources and entities, all implemented as elements of a computer program. When the program executes, these elements "model" the operation of the "real" system. The goal of such a model is to allow the user to study the dynamic behavior of the system; usually in an attempt to improve, in some manner, the underlying system.

It is natural to view most simulation models as collections of software components. Thus, designing simulation models as collections of objects is a natural merging of object-oriented design with simulation models.

There are several approaches to designing and implementing simulation models of systems. The choice of which approach to use depends on the justifications and goals for building the model. In all cases, we assume the primary goal is to construct a model of a specific system, where the system could be a "real" system that already exits or a system that does not exist but is being studied for some reason (one being that the system is going to be built in the future).

This paper focuses on designing and implementing simulation models as computer programs. There are other approaches which allow modelers to implement models without directly creating a program. In other words, someone else has created the program, and the end user can use that program to design and implement their specific model. However, in either case (writing a program or using another program), someone had to create a computer program which is a realization of the simulation model.

3 C++/CSIM17

C++ is one programming language that has been used to realize object-oriented designs of computer software. C++ is derived from the C programming language which is widely known and has been in use for many years. Thus, there is a large base of programmers who can readily adopt C++. In addition, C++ is widely supported: C++ compilers are available for almost all computer systems, there are many books which describe C++

and its use, and there are tools which help programmers to design and implement applications using C++.

CSIM17 is a toolkit which C programmers use to implement process-oriented, discrete-event simulation models of systems. C++/CSIM17 is a similar toolkit which provides all of the features of CSIM17, but it targets C++ programmers. Furthermore, it facilitates the use of object-oriented methodologies in the design and implementation of these models.

The remainder of this paper presents C++/CSIM17 and gives some examples of models rendered in C++/CSIM17.

4 C++/CSIM17 PROCESSES AND CLASSES

CSIM17 implements process-oriented, discrete-event simulation models. In such a model, the active "entities" of the system are modeled as processes, and the resources of the system are modeled as either facilities or storages. Mailboxes and events are used to synchronize or coordinate the behavior of the processes.

C++/CSIM17 provides four C++ classes which correspond to these resources and synchronization features. These, along with processes are described in the following sections.

4.1 Processes

A CSIM17 process is an independent computing entity. In CSIM17, a process is a procedure which executes a create statement. Executing this create statement converts a standard C or C++ procedure into a process. Each CSIM17 process appears to execute in a manner which is independent of other processes. Furthermore, processes can appear to execute simultaneously in simulated time. By using these processes, the modeler is able to simulate the concurrent activities of the model of the underlying system.

When activated, a process remains active (executing) until it executes a hold statement, terminates, or executes a statement which causes it to wait for some condition in the model happen (i.e., for some condition to be satisfied). These conditions will be described in subsequent sections.

CSIM17 processes are not system level processes; rather they are implemented as part of the CSIM17 runtime environment so as to execute in an efficient manner. A CSIM17 process preserves the variables which are local to the process (procedure) across periods of inactivity. In addition, CSIM17 processes can be invoked with input parameters, which are passed as function arguments.

4.2 The Facility Class

Facilities are used to model many of the components of the underlying system. A facility models a component as a collection of one or more "servers" and a single queue. A facility object is created by invoking the facility class constructor method. As with most C++ objects, the programmer can choose to create either static or dynamic objects. In C++/CSIM17, it is common to use global, static facility objects. For example, an assembly station could be established as follows:

```
facility assembly_station("assmbly");
```

This creates a facility object known to the program as assembly_station and with the string name "assmbly". An assembly_station with a single queue and three parallel stations (a multi-server facility in CSIM17 terminology) would be established as follows:

```
facility_ms assembly_station("assmbly", 3);
```

Processes can use a facility for a specified period of time by executing the *use* method in the facility class, as follows:

```
assembly_station.use(10.0);
```

When a process invokes the method

```
assembly_station.use(10.0),
```

the status of the server(s) in the assembly station is tested. If a "free" server is found, that server is assigned to the process and the process is delayed for 10.0 units of simulated time. If no "free" server is found, then the process is "suspended" (made inactive). At some later time, another process will finish it usage of a server and will release that server. When this happens, the server will be assigned to another process which was waiting for a free server. That suspend process will then be activated, enter its usage interval and then continue.

In addition to invoking the use() method, a process can reserve a server in a facility (invoke the reserve method); as soon as a server is reserved, the process can continue, either executing a hold statement or any other statement. Executing the sequence of statements:

```
assembly_station.reserve();
hold (5.0),
assembly_station.release ()
```

is equivalent to executing a single statement

```
assembly_station.use (5.0).
```

4.3 The Storage Class

A CSIM17 storage object is established with an amount of simulated storage. A process can then *allocate* a part of this storage. The allocate succeeds when the requested amount of storage is available; if the requested amount of storage is not available, the process is suspended until other processes have *deallocated* enough storage for the suspended request to be satisfied. A storage object named partition with 100 units of storage is established by invoking the storage class constructor method, as follows:

```
storage partition("partition", 100);
```

A process could allocate five partitions by invoking the *allocate* method, as follows:

```
partition.allocate(5);
```

4.4 The Mailbox Class

A mailbox object can be used to communicate messages between processes. A process can send a message to a mailbox, and some other process can receive that message from the mailbox. Messages are either a single-valued number or a pointer to a dynamic object or structure. Each mailbox object consists of two queues: one is the queue of unreceived messages and the other is the queue of processes waiting to receive a message.

In C++/CSIM17, using a combination of user-defined messages (classes) and mailbox objects allows the model builder to design and implement a variety of powerful, yet flexible inter-process communication mechanisms. As an example, assume that the programmer has already defined a class named item_c, which embodies the relevant features of an item to be assembled. Then the following statements could be used to generate an instance of item_c and then send it to the first assembly station for initial processing; the first step is establish a mailbox:

```
mailbox assembly_station_in("assmbly in");
```

Then, in the generator process, the following statements could be used:

```
item_c *item;
item = new item(...);
assembly_station_in.send((long) item);
```

Then, in the assembly_station process, the following statements would be used to receive this item:

```
item_c *next_item;
assembly_station_in.receive((long*)
    &next_item);
```

Presumably, the methods of item_c could then be used to extract the features of next_item and to modify its state before passing it on to the next station in the assembly process.

4.5 The Event Class

Event objects can be used to synchronize the activities of processes. An event is implemented as a two state variable with a queue for processes waiting for the event to "occur". A process "tests" the state of the event by executing either the *wait* method or the *queue* method. If the event object is the "not occurred" state, all of the of the processes testing the event will be suspended. When some other process invokes the *set* method (putting the event into the "occurred" state"), all of the waiting processes and one of the queued processes will be reactivated.

4.6 Other Classes and Procedures

C++/CSIM17 has some other classes which are used to complete the implementation of each model. These include classes for collecting data (*table, histogram, qtable and qhistogram*) and classes for generating streams of random numbers (*stream*). In addition, there are a set of procedures for generating either an overall *report*, or individualized subsets of reports. There are also routines for controlling the execution of the model and other aspects of its operation.

5 OBJECT REFINEMENT

The object-oriented characteristics of C++ and C++/CSIM17 can help modelers improve or "refine" the structure of a model as development and implementation proceed. This can be illustrated via an example. Assume that the facility assembly_station is as declared above. Furthermore, assume that an item process named "part" contains the statement

```
assembly_station.use(100.0);
```

meaning that "part" will use the assembly station for 100.0 units of time. At some later stage of the development of this model, someone decides that the model should have a more accurate and more detailed representation of this assembly process. In other words, describing the operation of this assembly station by simply allowing a specified amount of simulated time to pass is not good enough.

One way to "improve" the representation of this assembly station is to create a new assembly station which more accurately reflects the structure and operation of the "real" assembly station. For the sake of this example, assume that the real assembly station has two machines and that 40 % of the parts require processing at only the first station while the remaining 60 % of the parts require processing at both stations. Furthermore, the amount of processing time required is split in a "uniform" way over the two machines. The complete C++/CSIM17 implementation of this revised version is as follows:

```
class assembly_station_c {
private:
  facility *machine1;
  facility *machine2;
public:
  assembly_station_c(char* nm)
    { machine1 = new facility(nm);
      machine2 = new facility(nm);}
  ~assembly_station_c()
    {delete machine1; delete machine2;}
  void use(TIME t);
};

void assembly_station_c::use(TIME t)
{
  TIME t1;
  if(prob() <= 0.40) {
    machine1->use(t);
    }
  else {
    t1 = uniform(0.0, t);
    machine1->use(t1);
    machine2->use(t - t1);
    }
}
```

The three methods in the above class description are the constructor method, the destructor method and the use() method. As each instance of assembly_station_c is created, the constructor methods creates two dynamic facilities (machine1 and machine2). The destructor method (~assembly_station_c) deletes these two facilities when an object is destroyed. The use() method implements the more accurate processing pattern demanded by the modeler. In order to make use of this improved assembly station, the modeler is to make just one change to the program; the

```
facility assembly_station("assmbly")
```

is replaced with

```
assembly_station_c assembly("assmbly").
```

6 OBJECT INHERITANCE

C++ allows one class to "inherit" another class. An example of this in C++/CSIM17, consider the need to create a new kind of facility which collects some detailed data about response times. This new facility could be implemented as a regular facility with a CSIM17 table added. Then, when a request for service is completed, the response time could be automatically recorded in the table. The statements for accomplishing this are as follows:

```
class facility_data : public facility {
private:
  table *resp_tm;
public:
  facility_data(char* nm) : facility(nm) {
    resp_tm = new table("fac resp tm");}
  ~facility_data()   { delete resp_tm;}
  void use(TIME t);
};

void facility_data::use(TIME t)
{
  TIME t1;
  t1 = clock;
  facility::use(t);
  resp_tm->record(clock - t1);
}
```

In this example, the class named facility_data is derived from the base class facility. All of the methods of the facility class are available to the derived class with the exception of the "use()" method; the new class defines a new "use()" method.. This use of inheritance offers a convenient way of modifying the behavior of a base class.

7 SUMMARY

C++/CSIM17 is a powerful toolkit which allows simulation model builders to construct accurate, efficient models of their systems, while gaining the benefits of using object-oriented tools and methodologies. Because these models are written using the C++ programming language, programmers can quickly learn the CSIM17 syntax and features and can be designing and implementing process-oriented simulation models quickly. In addition, C++/CSIM17 is supported on many different system platforms, so moving models to different systems is very straightforward; most models can be recompiled and relinked on a new platform with no changes to the program. These and the many other features of C++/CSIM17 make this toolkit a good choice when accurate models of large, complex systems are needed.

ACKNOWLEDGMENTS

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

REFERENCES

Schwetman, H. 1986. CSIM: A C-based, process-oriented simulation language. In *Proceedings of the 1986 Winter Simulation Conference*, ed. J. Wilson, J. Henriksen, and S. Roberts, 387 - 396. Washington, DC.

Schwetman, H. 1988. Using CSIM to model complex systems. In *Proceedings of the 1988 Winter Simulation Conference*, ed. M. Abrams, P. Haigh, and J. Comfort, 246 - 253. San Diego, CA.

Schwetman, H. 1990. Introduction to process-oriented simulation and CSIM. In *Proceedings of the 1990 Winter Simulation Conference*, ed. O. Balci, R. Sadowski, and R. Nance, 154 - 157. New Orleans, LA.

Schwetman, H. 1994. CSIM17: A simulation model-building toolkit. In *Proceedings of the 1994 Winter Simulation Conference*, ed. J. Tew, S. Manivannan, D. Sadowski, A. Seila, 464 - 470. Orlando, FL.

AUTHOR BIOGRAPHY

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