Getting Started with CSIM for Java

For Java Programmers

Introduction

CSIM for Java™ is a library of Java classes and routines that give Java programmers the functionality of the CSIM library for discrete event simulations. This document is a tutorial for Java programmers who are not familiar with the CSIM toolkit. A separate document is available for C/C++ programmers who are new to Java, but are familiar with CSIM 19.

The M/M/1 Queue

The M/M/1 queue is a model of a system with one server and an unbounded queue for waiting customers. In a *single server* queue such as the M/M/1, there is a sequence of customers, with each customer arriving at successive points in time. An arriving customer requests use of the server. If the server is not busy, the arriving customer gets the server, allows a "service interval (of time)" to pass, releases the server, and departs as a completed transaction. If the server is busy (being used by another, earlier-arriving customer), the newly arriving customer joins the queue for waiting customers. When a customer releases the server and departs, the queue of waiting customers is checked; the customer at the head of the queue, if there is one, is given use of the server and removed from the queue.

The key parameters for this system are the time intervals between arriving customers and service time intervals for each customer. In most systems, the interarrival intervals are unequal (varying), as are the service intervals. One way to describe varying intervals is to say that they are described by a probability distribution. A common probability distribution that describes these kinds of varying intervals is the negative exponential distribution.

The notation "M/M/1 queue" specifies a single server queue (the 1) with interarrival intervals from a negative exponential distribution (the first M) and with service intervals from a negative exponential distribution (the second M).

Using CSIM for Java, a simulation model of an M/M/1 queue can be constructed with a *facility* (a simulated resource with a single server and a queue for waiting customers) and a sequence of processes that mimic the behavior of the sequence of arriving customers.

An M/M/1 Queue in CSIM for Java

This section describes how to model an M/M/1 queue in CSIM for Java. The example is a Java program named `App`. You will need to import the following files for this example:

```
import com.mesquite.csim.*;
import com.mesquite.csim.Process;
import com.mesquite.csim.file.Files;
import java.io.*;
```

The class *App* includes the required *main()* method; *App* extends the *Model* class, which is the basis for each CSIM for Java model. In this example, we set up a *PrintStream*, and assign the CSIM for Java output to this file. The *run* method for *App* uses the *start* method, to invoke the first process in the model. By convention, this process is named *sim*. The *App.start()* method should be called only one time per invocation of the *App* object.

The code for the *App* class is as follows:

```
public class App extends Model {
   public static void main(String args[]) {
        App model = new App();
        m_s = Files.Setfile("App.out");
        model.setOutputStream(m_s);
        model.run();
        model.report();
   }
   public App() {
        super("App");
   }
   public void run() {
        start(new Sim());
   }

   private static final double simTime = 10000.0;
   private static final double iarTime = 2.0;
   private static final double srvTime = 1.0;
   private FCFSFacility m_fac;
   private static PrintStream m_s;

   ....// processes (see below)

}
```

CSIM for Java uses Java threads as processes. Most of the details of dealing with *threads* and *threadGroups* are handled by the *model* class and the *Process* class.

The variables and constants for this model include the model parameters (*simTime*, *iarTime* and *srvTime*) plus the declarations for the facility and the *PrintStream* objects.

The remainder of the model consists of three processes:

- *Sim* – controls the execution of the model
- *Gen* – generates the arriving customers (jobs), and
- *Job* – represents the individual entities "using" the server at the facility.

The *Sim* process appears as follows:

```
   private class Sim extends Process {
        public Sim() {
             super("Sim");
```

```
        }
        public void run() {
              m_fac = new FCFSFacility("fac", 1);
              add(new Gen());
              hold(simTime);
        }
  }
```

The constructor calls the *Process* constructor using the *super()* statement. The *run* method is called by the thread package when the thread begins execution. In this example, the *Sim* method instantiates the facility, invokes the *Gen* process (using the *add()* statement) and holds for the duration of the model *(hold(simTime))*. The hold(t) statement allows t units of simulated time to pass for the process executing the statement. While one process is in a hold state, other processes are "executing". Hold statements specify the management of simulated time in the model. Notice that in CSIM for Java, the scheduling discipline for the facility is specified by the type of the facility. In this example, the scheduling discipline for *m_fac* is "first come, first served" (FCFS).

The *Gen* process appears as follows:

```
  private class Gen extends Process {
        public Gen() {
              super("Gen");
        }
        public void run() {
              while(true) {
                    add(new Job());
                    hold(rand.exponential(iarTime));
              }
        }
  }
```

In the *Gen* process, the *run* method executes "forever" (really until the model terminates). During each iteration of the While(true)-loop, the *Gen* process invokes a *Job* process and then holds for an interval of time specified by the *rand.exponential(iarTime)* function. Every time this function is called, a different value is returned. The values of this particular random number function appear to be drawn from a negative exponential probability distribution with mean *iarTime*.

The *Job* process appears as follows:

```
  private class Job extends Process {
        public Job() {
              super("Job");
        }
        public void run() {
              m_fac.use(rand.exponential(srvTime));
        }
  }
```

In the *Job* process, the *run* method calls the *use()* method for the *m_fac* object. The *use(t)* method implements the activities of the customer using the server as described above. In more detail, the *m_fac.use(t)* method performs the following actions:

- Test the status of the server. If the server is idle, the server is assigned to this instance of the *Job* process and delays (holds) for an interval of time (the service interval). At the end of this service interval, the server is released (made idle again). The queue of waiting processes is checked to see if another process is waiting to gain use of the server.
- If the server is busy, the process is placed in the queue of waiting processes and suspended. When this process gets to the first position in the queue of waiting processes (the head of the queue), it will be given use of the server once the server is available (e.g. when it is released by another instance of the *job* process).
- The *rand.exponential(srvTime)* argument generates a value which will be the service interval. These intervals appear to be drawn from a negative exponential probability distribution with mean specified by the constant *srvTime*.

The output for this model is generated when the *model.report()* method is called:

```
                    CSIM/Java Simulation Report

                   March 27, 2005 5:05:42 PM CST

              Ending Simulation time:          10000.000
              Elapsed Simulation time:         10000.000
              Execution (CPU) time:                0.831

FACILITY SUMMARY

facility     service    service          through-    queue     response   compl
name           disc      time    util.     put       length       time    count
-------------------------------------------------------------------------------
fac          fcfs       1.00954  0.512    0.50690    1.02003    2.01229     5069
```

A few notes about this CSIM for Java model:

- A CSIM for Java process extends the *Process* class. There can be multiple Process objects active at the same time. Parallel execution of these processes is simulated by these active processes.
- Simulated time passes when processes execute *hold(t)* statements.
- When an active process invokes another process (using the *add* method), the newly invoked process is set up and made ready to execute "now" in simulated time, and the active (invoking) process continues execution.
- Processes become active when some other process suspends itself. A process suspends itself when it executes a *hold(t)* statement or when it performs an action that could cause it to "wait" for another process to release a server.
- A process terminates when its *run* method ends.

- Simulated time advances in unequal increments. The runtime environment maintains a list of processes that will become active at some point in the future. The simulated clock advances to each of these process activation points.
- The *use(t)* method, for the *FCFSFacility* class, is a "macro" operation, based on the *reserve()* method, the *hold(t)*, method and the *release()* method.
- The *FCFSFacility* automatically collects statistics on the use of the facility and its server by the sequence of *Job* processes. These statistics are reported by the *model.report()* method:
  - *service time* – average of the sequence of service intervals
  - *util* – the percentage of the elapsed time that the server is busy
  - *throughput* – number of jobs completed per unit time
  - *queue length* – the average number of processes either waiting to gain access to the server or that are using the server
  - *response time* – the average amount that each process spends at the facility (both waiting to gain access the server and using the server)
  - *compl count* – the number of departures of completed jobs

The complete listing of the App module is as follows:

```
// Generic application: App.java

import com.mesquite.csim.*;
import com.mesquite.csim.Process;
import com.mesquite.csim.file.Files;
import java.io.*;

public class App extends Model {
   public static void main(String args[]) {
      App model = new App();
      m_s = Files.Setfile("App.out");
      model.setOutputStream(m_s);
      model.run();
      model.report();
   }
   public App() {
      super("App");
   }
   public void run() {
      start(new Sim());
   }

   private static final double simTime = 10000.0;
   private static final double iarTime = 2.0;
   private static final double srvTime = 1.0;
   private FCFSFacility m_fac;
   private static PrintStream m_s;
```

```
   private class Sim extends Process {
      public Sim() {
         super("Sim");
      }
      public void run() {
         m_fac = new FCFSFacility("fac", 1);
         add(new Gen());
         hold(simTime);
      }
   }

   private class Gen extends Process {
      public Gen() {
         super("Gen");
      }
      public void run() {
         while(true) {
            add(new Job());
            hold(rand.exponential(iarTime));
         }
      }
   }

   private class Job extends Process {
      public Job() {
         super("Job");
      }
      public void run() {
         m_fac.use(rand.exponential(srvTime));
      }
   }

}
```

Other CSIM for Java Features

CSIM for Java also includes a set of simulated resources:

- Single server and multi-server *facilities*
- *Storages*, and
- *Buffers*.

In addition, there are *events*, to synchronize the actions of processes, and *mailboxes*, to implement interprocess communication.

CSIM also includes objects to collect statistics, such as *tables*, *qtables*, *meters* and *boxes*. These objects can have histograms and confidence intervals.

And, there are several different functions for generating values from probability distributions. All of these features are described in detail in the *CSIM for Java User's Guide*.

CSIM for Java is distributed as a Java archive (jar) file. When a module is compiled with this jar file, the resulting class can be executed to yield the outputs as described.