Getting Started with CSIM for Java

For CSIM C/C++ Users

Introduction

CSIM for Java is a library of Java™ classes and routines to give Java programmers the functionality of the CSIM library for discrete event simulations, while mimicking the style of CSIM models. This document is a tutorial for programmers who are familiar with CSIM and are moving to the Java programming language. A separate document is available for Java programmers who are not already familiar with CSIM.

Example

The M/M/1 queue is a model of a basic system. A CSIM version of an M/M/1 queue is included as an appendix. In the Java version, the model is a class that extends the class *Model*. All of the necessary definitions are imported using the import statements:

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

In this example, the class that is the model is named *App* (and the file containing this class definition is `App.java`). The class *App* includes the required *main()* method.

```
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 in the model (analogous to CSIM processes). Most of the details of dealing with *threads* and *threadGroups* are handled by the *model* class and the *Process* class.

The *main()* method creates an instance of *App* named *model*. The *main()* method then initializes a *PrintStream* named *m_s*, and then makes *m_s* the *OutputStream* for the model. The model calls its *run()* method. *Note*: The file containing *App* must be named `App.java`, and the *main()* method must be in this class.

The *App* constructor calls its base class (*Model*) using the *super* statement. The *App.run* method starts the *Sim* process using the *App.start()* method. The *App.start()* method should be called only once per invocation of an *App* object.

The global variables include the model parameters (*simTime*, *iarTime* and *srvTime*) plus the declarations for the facility and the *PrintStream* objects. By making *m_s* a globally accessible object, the processes can add information to the output file, and by making *m_fac* globally accessible, all of the processes of the model can access the facility.

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))*.

Note: In CSIM for Java, the scheduling discipline for the facility is specified by the type of the facility (unlike CSIM, which calls the *set_servicefunc()* function to change the scheduling discipline).

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 exponentially distributed interarrival interval (the mean interval is specified by the value of *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. This *use* method operates in exactly the same manner as the *use()* method for a CSIM process. The stream of random numbers is the *rand* stream and the distribution of the service intervals is an exponential function with mean *srvTime*.

The output for this model is as follows:

```
                        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 CSIM programmer will find most of the familiar structures, features, etc. from the C/C++ version in CSIM for Java; in particular:

- processes
- facilities (see above)
- storages
- buffers
- events and event-sets
- mailboxes
- tables and qtables
- meters and boxes
- a subset of the probability distributions
- reports
- model control, including reset

Because CSIM for Java is a Java application, models can be developed and executed on any system with the Java Development Kit and Java Runtime Environment installed.

The *User's Guide,* available at www.mesquite.com/documentation, has a complete description of all of the structures and features in CSIM for Java.

Appendix: CSIM Version of M/M/1 Queue

```
// Example for CSIM/Java

#include "cpp.h"
#include <stdio.h>

const double simTime = 10000.0;
const double iarTime = 2.0;
const double srvTime = 1.0;

facility *m_fac;
FILE *s;

void gen();
void job();

extern "C" void sim()
{
   create("sim");
   s = fopen("java1.out", "w");
   set_output_file(s);
   m_fac = new facility("fac");
   gen();
   hold(simTime);
   report();
}

void gen()
{
   create("gen");
   while(true) {
      job();
      hold(exponential(iarTime));
   }
}

void job()
{
   create("job");
   m_fac->use(exponential(srvTime));
}
```

The output for this model is as follows:

```
        C++/CSIM Simulation Report (Version 19.0 for MS Visual C/C++)

                        Sun Mar 27 17:12:55 2005


                    Ending simulation time:     10000.000
                    Elapsed simulation time:    10000.000
                    CPU time used (seconds):        0.040


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
```