SSJ

A Java Library for Stochastic Simulation

User's Guide

Version: January 15, 2003

Pierre L'Ecuyer [1]

Département d'Informatique et de Recherche Opérationnelle
Université de Montréal

SSJ (an acronym for *Stochastic Simulation in Java*) is a library of classes, implemented in the Java programming language, offering general-purpose facilities for simulation programming. It supports the event view, process view, continuous simulation, and arbitrary mixtures of these.

[1]Most of the implementation of SSJ was done by Lakhdar Meliani for his master's thesis.

Contents

1 Introduction

SSJ is a Java library for simulation programming. It has been designed primarily for discrete-event stochastic simulations [1, 4, 9], but it also supports continuous and mixed simulation. Some ideas in the design of SSJ are inherited from the packages DEMOS [2] (based on the *Simula* language), SIMPascal [10] (based on the *Pascal* language), and SIMOD [14, 11] (based on the *Modula-2* language), among others.

Simulation models can be programmed in a variety of languages, including general-purpose programming languages such as FORTRAN, C, C++, Java, etc., specialized languages such as GPSS, SIMAN, SLAM, SIMSCRIPT, etc., and graphical simulation environments such as Arena, Automod, etc. [1].

Specialized languages and environments provide higher-level tools but usually at the expense of being less flexible than general purpose programming languages. In the popular simulation languages, one must frequently turn down to general-purpose languages (such as FORTRAN, C, or VisualBasic, for example) to program complex aspects of a model and operations not supported by the specialized language. Compilers and supporting environments for specialized languages are less widely available and cost more than for general purpose languages. Another obstacle to using a specialized language: one must learn it. This is a non-negligible time investment, especially for an occasional use, given that these languages have their own (sometimes eccentric) syntax and semantic. Some high-level simulation environments propose a "no-programming" approach, where models are specified by manipulating graphical objects on the computer screen by point, click, drag, and drop operations, as in computer games. This approach is very convenient for building models that happen to fit in the frameworks pre-programmed in the software. But for large real-life systems, such nice fits are more the exception than the rule.

SSJ is implemented as a collection of classes in the Java language. It provides convenient tools for simulation programming without giving away the generality and the power provided by a widely-used general-purpose programming language. These predefined classes contain facilities for generating random numbers for various distributions, collecting statistics, managing a simulation clock and a list of future events, synchronizing the interaction between simulated concurrent processes, etc. SSJ supports both the event view and the process view, as well as continuous simulation (where some variables evolve according to differential equations), and the three can be combined.

In the process-oriented paradigm, *active objects* in the system, called *processes*, have a method that describe their behavior in time. The processes can interact, can be suspended and reactivated, can be waiting for a given resource or a given condition, can be created and destroyed, etc. These processes may represent more or less autonomous objects such as machines and robots in a factory, customers in a restaurant, vehicles in a transportation network, etc. Process-oriented programming is a natural way of describing a complex system [3, 9, 8]. On the other hand, other systems are more convenient to model simply with *events*, that are executed instantaneously in the simulation time frame. The use of events is sometimes preferred because it gives a faster simulation program, by avoiding the process-synchronization overhead. Events and processes can be mixed freely in SSJ.

The next section gives an overview of SSJ. Section 3 provides a series of simulation programs written in Java using SSJ. These programs are explained in details and illustrate the use of SSJ. Studying these examples is probably the best way to start becoming familiar with the tools of SSJ. A complete documentation of all the classes provided by SSJ is given in Appendix A. The reader can refer to this appendix while (and after) studying the examples.

2 An overview of SSJ

This section gives a quick overview of the packages and classes of SSJ. A detailed description is given in Appendix A. One can easily extend these facilities by adding more classes, and we intend to do it in the future.

The interface `RandomStream` and its implementations provide the basic random number generator (RNG) facilities. Each implementation of this interface implements a different type of uniform RNG, with multiple streams and substreams. Only the `RandMrg` implementation is available for the moment, but others will be available later. The classes `Rand1`, `Rand2`, `Ranlib`, etc. provide methods for generating non-uniform random variates from several kinds of distributions.

`StatProbe` and its subclasses `Tally` and `Accumulate` provide elementary tools for collecting statistics and computing confidence intervals. The class `List` implements *doubly linked* lists, with tools for inserting, removing, and viewing objects in the list, and automatic statistical collection. These list can contain any kind of `Object`.

The scheduling part of discrete-event simulations is managed by the "chief-executive" class `Sim`, which contains the simulation clock and the central monitor. The event list is taken from one of the implementations of the interface `EventList`, which provide different kinds of event list implementations. One can change the event list implementation via the method `Sim.init`. The default implementation (and the only one available for the moment) is `DoublyLinked`. The classes `Event` and `Process` provide the facilities for creating and scheduling events and processes in the simulation. Each type of event or process must be defined by defining a class that extends `Event` or `Process`. The class `Continuous` provide tools for continuous simulation, where certain variables vary continuously according to ordinary differential equations.

The classes `Resource`, `Bin`, and `Condition`, provide additional mechanisms for process synchronization. A `Resource` corresponds to a facility with limited capacity and a waiting queue. A `Process` can request an arbitrary number of units of a `Resource`, may have to wait until enough units are available, can use the `Resource` for a certain time, and eventually releases it. A `Bin` allows producer/consumer relationships between processes. It corresponds essentially to a pile of free tokens and a queue of processes waiting for the tokens. A *producer* adds tokens to the pile whereas a *consumer* (a process) can ask for tokens. When not enough tokens are available, the consumer is blocked and placed in the queue. The class `Condition` supports the concept of processes waiting for a certain boolean condition to be true before continuing their execution.

3 Exemples of simulation programs using SSJ

In this section we present some examples of SSJ programs (i.e., simulation programs in Java using SSJ). Studying these examples is a good way to learn how to use SSJ. The reader can refer at need to the functional definitions of the SSJ classes and methods, in Appendix A.

In Section 3.1, we start with a very simple classical example: a single queue. We give different variants of this example, illustrating the mixture of processes and events. Section 3.2 gives a small example of a deterministic continuous simulation. In Sections 3.3 and 3.4, we give examples of a jobshop model and a time-shared computer model, adapted from [9]. A queueing model of a bank, taken from [4], is programmed in Section 3.5, with both the process view and the event view. In Section 3.6, we simulate a model of guided tours for groups of people, where the process synchronization is slightly more complicated than for the earlier models.

3.1 A single-server queue

Our first example is a *single-server queue*, where customers arrive randomly and are served one by one in their order of arrival, i.e., *first in, first out* (FIFO). We suppose that the times between successive arrivals are exponential random variables with mean 10 minutes, that the service times are exponential random variables with mean 9 minutes, and that all these random variables are mutually independent. The customers arriving while the server is busy must join the queue. The system initially starts empty. We want to simulate the first 1000 hours of its operation and compute statistics such as the mean waiting time per customer, the mean queue length, etc.

This simple model is well-known in queueing theory: It is called an $M/M/1$ queue. Simple formulas are available for this model to compute the average waiting time per customer, average queue length, etc., over an *infinite* time horizon [7]. When the time horizon is finite, these expectations can also be computed by numerical methods. The fact that we use this model to give a first tasting of SSJ should not be interpreted to mean that simulation is necessarily the best tool for it.

We give three examples of simulation programs for the $M/M/1$ queue: The first one is event-oriented, the second one is process-oriented, and the third one uses a simple recurrence. The first program is longer and more complicated than the other two; it shows how things work at a lower level.

3.1.1 Event-oriented program

Figure 1 gives an event-oriented simulation program, where a subclass of the class `Event` is defined for each type of event that can occur in the simulation: arrival of a customer (`Arrival`), departure of a customer (`Departure`), and end of the simulation (`EndOfSim`). Each event *instance* is inserted into the *event list* upon its creation, with a scheduled time of occurence, and is *executed* when the simulation clock reaches this time. Executing an event means invoking its `actions` method. Each event subclass must implement this method. The simulation clock and the event list (i.e., the list of events scheduled to occur in the future) are maintained behind the scenes by the class `Sim` of SSJ.

When `QueueEv` is instantiated by the `main` method, the program creates two streams of random numbers, two lists, and two statistical probes (or collectors). The random number streams `genArr` and `genServ` can be viewed as virtual random number generators that generate the times between successive arrivals and the service times, respectively. The lists `waitList` and `servList` contain the customers waiting in the queue and the customer in service (if any), respectively. Maintaining a list for the customer in service may seem exaggerated, because this list never contains more than one object, but the current design has the advantage of working with very little change if the queueing model has more than one server, and in other more general situations.

The statistical probe `custWaits` collects statistics on the customer's waiting times. It is of the class `Tally`, which is appropriate when the statistical data of interest is a sequence of observations $X_1, X_2, \ldots$ of which we might want to compute the sample mean, variance, and

```
public class QueueEv {

    static final double meanArr     = 10.0;
    static final double meanServ    =  9.0;
    static final double timeHorizon = 1000.0;

    RandMrg genArr     = new RandMrg ();
    RandMrg genServ    = new RandMrg ();
    List    waitList   = new List ("Customers waiting in queue");
    List    servList   = new List ("Customers in service");
    Tally   custWaits  = new Tally ("Waiting times");
    Accumulate totWait = new Accumulate ("Size of queue");

    class Customer { double arrivTime, servTime; }

    public static void main (String[] args) { new QueueEv(); }

    public QueueEv() {
        Sim.init();
        new EndOfSim().schedule (timeHorizon);
        new Arrival().schedule (Rand1.expon (genArr, meanArr));
        Sim.start();
        }
    class Arrival extends Event {
        public void actions() {
            new Arrival().schedule (Rand1.expon (genArr, meanArr));
                                          // The next arrival.
            Customer cust = new Customer();  // Cust just arrived.
            cust.arrivTime = Sim.time();
            cust.servTime = Rand1.expon (genServ, meanServ);
            if (servList.size() > 0) {       // Must join the queue.
                waitList.insert (cust, List.LAST);
                totWait.update (waitList.size());
            } else {                         // Starts service.
                servList.insert (cust, List.LAST);
                new Departure().schedule (cust.servTime);
                custWaits.update (0.0);
                }
            }
        }
    class Departure extends Event {
        public void actions () {
            servList.remove (List.FIRST);
            if (waitList.size () > 0) {
                // Starts service for next one in queue.
                Customer cust = (Customer) waitList.remove (List.FIRST);
                servList.insert (cust, List.LAST);
                new Departure().schedule (cust.servTime);
                custWaits.update (Sim.time () - cust.arrivTime);
                totWait.update (waitList.size ());
                }
            }
        }
    class EndOfSim extends Event {
        public void actions () {
            custWaits.report();   totWait.report();
            Sim.stop();
            }
        }
    }
```

Figure 1: Event-oriented simulation of an $M/M/1$ queue.

so on. Every update of a `Tally` probe brings a new observation X_i, which corresponds here to a customer's waiting time in the queue. A new observation is given to this probe by the `update` method each time a customer starts its service. The other statistical probe, `totWait`, is of the class `Accumulate`, which means that it computes the integral (and, eventually, the time-average) of a continuous-time stochastic process with piecewise-constant trajectory. Here, the stochastic process of interest is the length of the queue as a function of time. One must call `totWait.update` whenever there is a change in the queue size, to update the (hidden) *accumulator* that keeps the current value of the integral of the queue length. This integral is equal, after each update, to the total waiting time in the queue, for all the customers, since the beginning of the simulation.

Each customer is an object with two fields: `arrivTime` memorizes this customer's arrival time to the system, and `servTime` memorizes its service time. This object is created, and its fields are initialized, when the customer arrives.

The method `Sim.init`, invoked in the constructor `QueueEv`, initializes the clock and the event list, whereas `Sim.start` actually starts the simulation by advancing the clock to the time of the first event in the event list, removing this event from the list, and executing it. This is repeated until either `Sim.stop` is called or the event list becomes empty. `Sim.time` returns the current time on the simulation clock. Here, two events are scheduled before starting the simulation: The end of the simulation at time 1000, and the arrival of the first customer at a random time that has the exponential distribution with mean 10, generated using the random stream `genArr`. (The method `Rand1.expon (g, m)` returns an exponential random variable with mean `m`, generated with the random stream `g`.)

The method `actions` of the class `Arrival` describes what happens when an arrival occurs. Arrivals are scheduled by a domino effect: The first action of each arrival event is to schedule the next one in a random number of time units, generated from the exponential distribution with mean 10. Then, the newly arrived customer is created, its arrival time is set to the current simulation time, and its service time is generated from the exponential distribution with mean 9, using the random stream `genServ`. If the server is busy, this customer is inserted at the end of the queue (i.e., the list `waitList`) and the statistical probe `totWait`, that keeps track of the size of the queue, is updated. Otherwise, the customer is inserted in the server's list `servList`, its departure is scheduled to happen in a number of time units equal to its service time, and a new observation of 0.0 is given to the statistical probe `custWaits` that collects the waiting times.

When a `Departure` event occurs, the customer in service is removed from the list (and disappears). If the queue is not empty, the first customer is removed from the queue (`waitList`) and inserted in the server's list, and its departure is scheduled. The waiting time of that customer (the current time minus its arrival time) is given as a new observation to the probe `custWaits`, and the probe `totWait` is also updated with the new (reduced) size of the queue.

The event `EndOfSim` prints statistical reports for the two probes and stops the simulation. The program prints the results shown in Figure 2.

```
REPORT on Tally stat. collector ==> Waiting times
  min        max       average      standard dev   nb. obs.
   0       113.721      49.554         22.336         97

REPORT on Accumulate stat. collector ==> Size of queue

  From time  To time     Min        Max         Average
    0          1000       0          12          4.85
```

Figure 2: Results of the program `QueueEv`.

3.1.2 Process-oriented program

Typical simulation languages offer higher-level constructs than those used in the program of Figure 1, and so does SSJ. This is illustrated by our second implementation of the single-server queue model, in Figure 3, based on a paradigm called the *process-oriented* approach.

```
public class QueueProc {

    static final double meanArr     = 10.0;
    static final double meanServ    =  9.0;
    static final double timeHorizon = 1000.0;

    Resource server  = new Resource (1, "server");
    RandMrg  genArr  = new RandMrg ();
    RandMrg  genServ = new RandMrg ();

    public static void main (String[] args) {new QueueProc(); }

    public QueueProc () {
        Sim.init();
        server.collectStat (true);
        new EndOfSim().schedule (timeHorizon);
        new Customer().schedule (Rand1.expon (genArr, meanArr));
        Sim.start();
        }

    class Customer extends Process {
        public void actions () {
            new Customer().schedule (Rand1.expon (genArr, meanArr));
            server.request (1);
            delay (Rand1.expon (genServ, meanServ));
            server.release (1);
            }
        }

    class EndOfSim extends Event {
        public void actions () {
            server.report();
            Sim.stop();
            }
        }
    }
```

Figure 3: Process-oriented simulation of an $M/M/1$ queue.

In the event-oriented implementation, each customer was a *passive* object, storing two real numbers, and performing no action by itself. In the process-oriented implementation given in Figure 3, each customer (instance of the class `Customer`) is a process whose activities are described by its method `actions`. This is implemented by associating a Java `Thread` to each `Process`. The server is an object of the class `Resource`, created when `QueueProc` is instantiated by `main`. It is a *passive* object, in the sense that it executes no code. Active resources, when needed, can be implemented as processes.

When it starts executing its actions, a customer first schedules the arrival of the next customer, as in the event-oriented case. (Behind the scenes, this effectively schedules an event, in the event list, that will start a new customer instance. The class `Process` is a subclass of `Event`, which permits one to schedule processes just like events.) The customer then requests the server by invoking `server.request`. If the server is free, the customer gets it and can continue its execution immediately. Otherwise, the customer is automatically (behind the scenes) placed in the server's queue, is suspended, and resumes its execution only when it obtains the server. When its service can start, the customer invokes `delay` to freeze itself for a duration equal to its service time, which is again generated from the exponential distribution with mean 9 using the random stream `genServ`. After this delay has elapsed, the customer releases the server and ends its life. Invoking `delay(d)` can be interpreted as scheduling an event that will resume the execution of the process in `d` units of time. Note that several distinct customers can co-exist in the simulation at any given point in time, and be at different phases of their `actions` method.

The constructor `QueueProc` initializes the simulation, invokes `collectStat` to specify that detailed statistical collection must be performed automatically for the resource `server`, schedules an event `EndOfSim` at time 1000, schedules the first customer's arrival, and starts the simulation. The `EndOfSim` event prints a detailed statistical report on the resource `server`.

It should be pointed out that in the `QueueProc` program, the service time of a customer is generated only when the customer starts its service, whereas for `QueueEv`, it was generated at customer's arrival. For this particular model, it turns out that this makes no difference in the results, because the customers are served in a FIFO order and because one random number stream is dedicated to the generation of service times. However, this may have considerable impact in other situations.

The process-oriented program here is more compact and more elegant than its event-oriented counterpart. This tends to be often true: Process-oriented programming frequently gives less cumbersome and better looking programs. On the other hand, the process-oriented implementations also tend to execute more slowly, because they involve more overhead. In fact, process management is done via the event list: processes are started, suspended, reactivated, and so on, by hidden events. If the execution speed of a simulation program is really important, it may be better to stick to an event-oriented implementation.

[2]

[2]From Pierre: À faire: comparaisons de vitesse d'exécution.

```
REPORT ON RESOURCE : Server
  From time : 0.0    to time : 1000.0
                 min          max      average    Std. Dev.  nb. obs.
  Capacity         1            1            1
  Utilisation      0            1        0.999
  Queue Size       0           12         4.85
  Wait             0      113.721       49.554       22.336        97
  Service      0.065       41.021       10.378       10.377        96
  Sojourn     12.828      124.884       60.251       21.352        96
```

Figure 4: Results of the program `QueueProc`.

Figure 4 shows the output of the program `QueueProc`. It contains more information than the output of `QueueEv`. It gives statistics on the server utilization, queue size, waiting times, service times, and sojourn times in the system. (The sojourn time of a customer is its waiting time plus its service time.)

We see that by time $T = 1000$, 97 customers have completed their waiting and 96 have completed their service. The maximal length of the queue has been 12 and its average length between time 0 and time 1000 was 4.85. The waiting times were in the range from 0 to 113.721, with an average of 49.554, while the service times were from 0.065 to 41.021, with an average of 10.378 (recall that the theoretical mean service time is 9.0). Clearly, the largest waiting time and largest service time belong to different customers. The average waiting and service times do not sum up to the average sojourn time because there is one more observation for the waiting times.

The report also gives the empirical standard deviations of the waiting, service, and sojourn times. It is important to note that these standard deviations should *not* be used to compute confidence intervals for the expected average waiting times or sojourn times in the standard way, because the observations here (e.g., the successive waiting times) are strongly dependent, and also not identically distributed. Appropriate techniques for computing confidence intervals in this type of situation are described, e.g., in [5, 9].

3.1.3 A simpler program based on Lindley's recurrence

The aim of the previous programs was to illustrate the notions of events and processes by a simple example. However, for a single-server queue, if W_i and S_i are the waiting time and service time of the ith customer, and A_i is the time between the arrivals of the ith and $(i+1)$th customers, we have $W_1 = 0$ and the W_i follow the recurrence

$$W_{i+1} = \max(0,\ W_i + S_i - A_i), \tag{1}$$

known as Lindley's equation [7]. The program of Figure 5 exploits (1) to compute the average waiting time of the first 100 customers. This is different than for the previous programs, because we now fix the total number of customers instead of fixing the time horizon.

```
public class QueueLindley {

    static final double meanArr  = 10.0;
    static final double meanServ =  9.0;
    static final int nbCust      = 100;

    RandMrg genArr  = new RandMrg ();
    RandMrg genServ = new RandMrg ();
    Tally custWaits = new Tally ("Customer's waiting times");

    public static void main (String[] args) { new QueueLindley(); }

    public QueueLindley() {
        double Wi = 0.0;
        custWaits.update (Wi);
        for (int i = 2; i <= nbCust; i++) {
            Wi += Rand1.expon (genServ, meanServ)
                  - Rand1.expon (genArr, meanArr);
            if (Wi < 0.0) Wi = 0.0;
            custWaits.update (Wi);
            }
        custWaits.report();
        }
    }
```

Figure 5: A simulation based on Lindley's recurrence

3.2 Continuous simulation: A prey-predator system

We consider a classical prey-predator system, where the preys are food for the predators (see, e.g., [9], page 87). Let $x(t)$ and $z(t)$ be the numbers of preys and predators at time t, respectively. These numbers are integers, but as an approximation, we shall assume that they are real-valued variables evolving according to the differential equations

$$\begin{aligned} x'(t) &= rx(t) - cx(t)z(t) \\ z'(t) &= -sz(t) + dx(t)z(t) \end{aligned}$$

with initial values $x(0) = x_0 > 0$ et $z(0) = z_0 > 0$. This system is actually a Lotka-Volterra system of differential equations, and has a known analytical solution. Here, in the program of Figure 6, we simply simulate its evolution, to illustrate the continuous simulation facilities of SSJ.

This program prints the triples $(t, x(t), z(t))$ at values of t that are multiples of `h`, one triple per line. This is done by an event of class `PrintPoint`, which is rescheduled at every `h` units of time. This output can be redirected to a file for later use, for example to plot a graph of the trajectory. The continuous variables `x` and `z` are instances of the classes `Preys` and `Preds`, whose method `deriv` give their derivative $x'(t)$ and $z'(t)$, respectively. The differential equations are integrated by a Runge-Kutta method of order 4.

```
public class PreyPred {

    double r = 0.005, c = 0.00001, s = 0.01, d = 0.000005, h = 5.0;
    double x0 = 2000.0, z0 = 150.0;
    double horizon = 501.0;
    Continuous x = new Preys ();
    Continuous z = new Preds ();

    public static void main (String[] args) { new PreyPred (); }

    public PreyPred () {
        Sim.init();
        new EndOfSim().schedule (horizon);
        new PrintPoint().schedule (h);
        Continuous.selectIntegMethod (Continuous.RUNGEKUTTA4, h);
        x.startInteg (x0);    z.startInteg (z0);
        Sim.start();
        }

    public class Preys extends Continuous {
        public double deriv (double t) {
           return (r * value() - c * value() * z.value());
           }
        }

    public class Preds extends Continuous {
        public double deriv (double t) {
           return (-s * value() + d * x.value() * value());
           }
        }

    class PrintPoint extends Event {
        public void actions () {
            System.out.println (Sim.time() + "  "
                + x.value() + "  " + z.value());
            this.schedule (h);
            }
        }

    class EndOfSim extends Event {
        public void actions () { Sim.stop(); }
        }
    }
```

Figure 6: Simulation of the prey-predator system

3.3 A jobshop model

This example is adapted from Law and Kelton [9], Section 2.6, and from [11]. A jobshop contains M groups of machines, the mth group having s_m identical machines, for $m = 1, \ldots, M$. It is modeled as a network of queues: each group has a single FIFO queue, with s_m identical servers for the mth group. There are N types of tasks arriving to the shop at random. Tasks of type n arrive according to a Poisson process with rate λ_n per hour, for $n = 1, \ldots, N$. Each type of task requires a fixed sequence of operations, where each operation must be performed on a specific type of machine and has a deterministic duration. A task of type n requires p_n operations, to be performed on machines $m_{n,1}, m_{n,2}, \ldots, m_{n,p_n}$, in that order, and whose respective durations are $d_{n,1}, d_{n,2}, \ldots, d_{n,p_n}$, in hours. A task can pass more than once on the same machine type, so p_n may exceed M.

We want to simulate the jobshop for T hours, assuming that it is initially empty, and start collecting statistics only after a warmup period of T_0 hours. We want to compute: (a) the average sojourn time in the shop for each type of task and (b) the average utilization rate, average length of the waiting queue, and average waiting time, for each type of machine, over the time interval $[T_0, T]$. For the average sojourn times and waiting times, the counted observations are the sojourn times and waits that *end* during the time interval $[T_0, T]$. Note that the only randomness in this model is in the task arrival process.

The program `Jobshop` in Figure 7 performs this simulation. Each group of machine is viewed as a resource, with capacity s_m for the group m. The different *types* of task are objects of the class `TaskType`. This class is used to store the parameters of the different types: their arrival rate, the number of operations, the machine type and duration for each operation, and a statistical collector for their sojourn times in the shop. (In the program, the machine types and task types are numbered from 0 to $M - 1$ and from 0 to $N - 1$, respectively, because array indices in Java start at 0.)

The tasks that circulate in the shop are objects of the class `Task`. The `actions` method in class `Task` describes the behavior of a task from its arrival until it exits the shop. Each task, upon arrival, schedules the arrival of the next task of the same type. The task then run through the list of its operations. For each operation, it requests the appropriate type of machine, keeps it for the duration of the operation, and releases it. When the task terminates, it sends its sojourn time as a new observation to the collector `statSojourn`.

Before starting the simulation, the class `Jobshop` first schedules two events: One for the end of the simulation and one for the end of the warmup period. The latter simply reinitializes the statistical collectors.

With this implementation, the event list always contain N "task arrival" events, one for each type of task. An alternative implementation would be that each task schedules another task arrival in a number of hours that is an exponential r.v. with mean $1/\lambda$, where $\lambda = \lambda_0 + \cdots + \lambda_{N-1}$ is the global arrival rate, and then the type of each arriving task is n with probability λ_n/λ, independently of the others. Initially, a *single* arrival would be scheduled by the class `Jobshop`. This approach is stochastically equivalent to the current implementation (see, e.g., [4, 16]), but the event list contains only one "task arrival" event at a time. On the other hand, there is the additional work of generating the task type on each arrival.

```
import java.io.*;
import java.util.*;

public class Jobshop {

   int nbMachTypes;           // Number of machine types M.
   int nbTaskTypes;           // Number of task types N.
   double warmupTime;         // Warmup time T_0.
   double horizonTime;        // Horizon length T.
   boolean warmupDone;        // Becomes true when warmup time is over.
   Resource[] machType;       // The machines groups as resources.
   TaskType[] taskType;       // The task types.
   RandomStream genArr = new RandMrg();   // RNG stream for arrivals.
   BufferedReader input;

   static public void main (String[] args) throws IOException {
      new Jobshop ();
      }

   public Jobshop () throws IOException {
      Sim.init ();
      readData ();
      warmupDone = false;
      endWarmup.schedule (warmupTime);
      endOfSim.schedule (horizonTime);
      for (int n=0; n < nbTaskTypes; n++) {
         new Task(taskType[n]).schedule (Rand1.expon (genArr,
               1.0 / taskType[n].arrivalRate));
         }
      Sim.start();
      }

   // Reads data file, and creates machine types and task types.
   void readData () throws IOException {
      input = new BufferedReader (new FileReader ("Jobshop.dat"));
      StringTokenizer line = new StringTokenizer (input.readLine ());
      warmupTime = Double.parseDouble (line.nextToken());
      line = new StringTokenizer (input.readLine ());
      horizonTime = Double.parseDouble (line.nextToken());
      line = new StringTokenizer (input.readLine ());
      nbMachTypes = Integer.parseInt (line.nextToken());
      nbTaskTypes = Integer.parseInt (line.nextToken());
      machType = new Resource[nbMachTypes];
      for (int m=0; m < nbMachTypes; m++) {
         line = new StringTokenizer (input.readLine ());
         String name = line.nextToken();
         int nb = Integer.parseInt (line.nextToken());
         machType[m] = new Resource (nb, name);
         }
      taskType = new TaskType[nbTaskTypes];
      for (int n=0; n < nbTaskTypes; n++) taskType[n] = new TaskType ();
      input.close ();
      }
```

Figure 7: A jobshop simulation

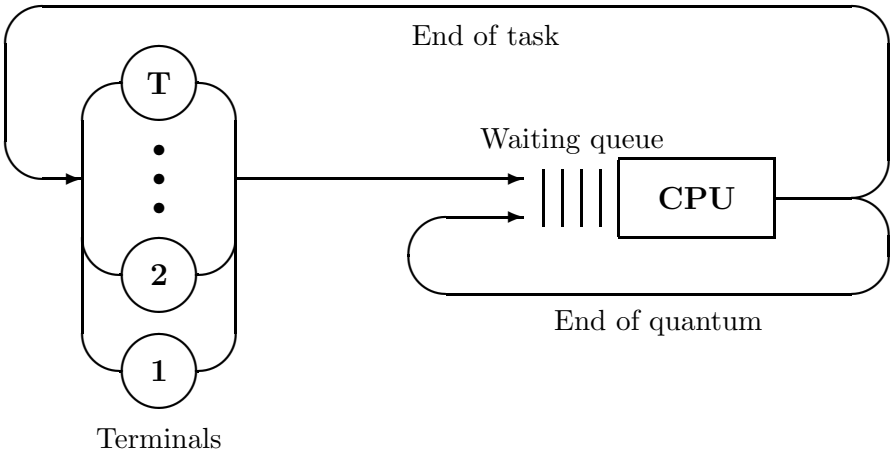
```
class TaskType {
   public String     name;        // Task name.
   public double     arrivalRate; // Arrival rate.
   public int        nbOper;      // Number of operations.
   public Resource[] machOper;    // Machines where operations occur.
   public double[]   lengthOper;  // Durations of operation.
   public Tally      statSojourn; // Stats on sojourn times.

   // Reads data for new task type and creates data structures.
   TaskType () throws IOException {
      StringTokenizer line = new StringTokenizer (input.readLine ());
      statSojourn = new Tally (name = line.nextToken());
      arrivalRate = Double.parseDouble (line.nextToken());
      nbOper = Integer.parseInt (line.nextToken());
      machOper = new Resource[nbOper];
      lengthOper = new double[nbOper];
      for (int i=0; i < nbOper; i++) {
         int p = Integer.parseInt (line.nextToken());
         machOper[i] = machType [p-1];
         lengthOper[i] = Double.parseDouble (line.nextToken());
         }
      }

   // Performs the operations of this task (to be called by a process).
   public void performTask (Process p) {
      double arrivalTime = Sim.time();
      for (int i=0; i < nbOper; i++) {
         machOper[i].request (1);
         p.delay (lengthOper[i]);
         machOper[i].release (1);
         }
      if (warmupDone) statSojourn.update (Sim.time() - arrivalTime);
      }
   }

public class Task extends Process {
   TaskType type;

   Task (TaskType typ) { type = typ; }

   public void actions () {
      new Task(type).schedule (Rand1.expon (genArr,
            1.0 / type.arrivalRate));
         type.performTask (this);
      }
   }

Event endWarmup = new Event () {
   public void actions () {
      for (int m=0; m < nbMachTypes; m++) machType[m].collectStat (true);
      warmupDone = true;
      }
   };

Event endOfSim = new Event () {
   public void actions () {
      for (int m=0; m < nbMachTypes; m++) machType[m].report();
      for (int n=0; n < nbTaskTypes; n++) taskType[n].statSojourn.report();
      Sim.stop ();   Process.killAll ();
      }
   };
}
```

Figure 7: A jobshop simulation (continued)

3.4 A time-shared computer system

This example is adapted from [9], Section 2.4. Consider a simplified time-shared computer system comprised of T identical and independent terminals, all busy, using a common server (e.g., for database requests, or central processing unit (CPU) consumption, etc.). Each terminal user sends a task to the server at some random time and waits for the response. After receiving the response, he thinks for some random time before submitting a new task, and so on.

We assume that the thinking time is an exponential random variable with mean μ, whereas the server's time needed for a request is a Weibull random variable with parameters α and λ. The tasks waiting for the server form a single queue with a *round robin* service policy with *quantum size* q, which operates as follows. When a tasks obtains the server, if it can be completed in less than q seconds, then it keeps the server until completion. Otherwise, it gets the server for q seconds and returns to the back of the queue to be continued later. In both cases, there is also h additional seconds of *overhead* for changing the task that has the server's attention.



The *response time* of a task is defined as the difference between the time when the task ends (including the overhead h at the end) and the arrival time of the task to the server. We are interested in the *mean response time*, in steady-state. We will simulate the system until N tasks have ended, with all terminals initially in the "thinking" state. To reduce the initial bias, we will start collecting statistics only after N_0 tasks have ended (so the first N_0 response times are not counted by our estimator, and we take the average response time for the $N - N_0$ response times that remain). This entire simulation is repeated R times, independently, so we can estimate the variance of our estimator.

Suppose we want to compare the mean response times for two different configurations of this system, where a configuration is characterized by the vector of parameters $(T, q, h, \mu, \alpha, \lambda)$. We will make R independent simulation runs (replications) for each configuration. To compare the two configurations, we want to use *common random numbers*, i.e., the same streams of random numbers across the two configurations. We couple the

simulation runs by pairs: for run number i, let R_{1i} and R_{2i} be the mean response times for configurations 1 and 2, and let

$$D_i = R_{1i} - R_{2i}.$$

We use the same random numbers to obtain R_{1i} and R_{2i}, for each i. The D_i are nevertheless independent random variables (under the blunt assumption that the random streams really produce independent uniform random variables) and we can use them to compute a confidence interval for the difference d between the theoretical mean response times of the two systems. Using common random numbers across R_{1i} and R_{2i} should reduce the variance of the D_i and the size of the confidence interval.

The program of Figure 8 performs this simulation. In `TimeShared`, the variable `conf` indicate the current configuration number. For each configuration, we make `nbRep` simulation runs. The array `meanConf1` memorizes the values of R_{1i}, and the statistical probe `statDiff` collect the differences D_i, in order to compute a confidence interval for d. After all the runs for the first configuration have been completed, the random number streams are reset to their initial seeds, so that the two configurations get the same random numbers. The random streams are also reset to the beginning of their next substream after each run, to make sure that for the corresponding runs for the two configurations, the generators start from exactly the same seeds and generate the same numbers.

For each simulation run, the statistical probe `meanInRep` is used to compute the average response time for the $N - N_0$ tasks that terminate after the warmup. It is initialized before each run and updated with a new observation at the ith task termination, for $i = N_0 + 1, \ldots, N$. At the beginning of a run, a `Terminal` process is activated for each terminal. When the Nth task terminates, the corresponding process invokes `Sim.stop` to stop the simulation. This kills all the `Terminal` processes and returns the control to the instruction that follows the call to `simulOneRun` in `TimeShared`.

For a concrete example, let $T = 20$, $h = .001$, $\mu = 5$ sec., $\alpha = 1/2$ and $\lambda = 1$ for the two configurations. With these parameters, the mean of the Weibull distribution is 2. Take $q = 0.1$ for configuration 1 and $q = 0.2$ for configuration 2. We also choose $N_0 = 100$, $N = 1100$, and $R = 10$ runs. With these numbers in the data file, the program gives the results of Figure 9. The confidence interval on the difference between the response time with $q = 0.1$ and that with $q = 0.2$ contains only positive numbers. We can therefore conclude that the mean response time is significantly shorter (statistically) with $q = 0.2$ than with $q = 0.1$ (assuming that we can neglect the bias due to the choice of the initial state). To gain better confidence in this conclusion, we could repeat the simulation with larger values of N_0 and N.

Of course, the model could be made more realistic by considering, for example, different types of terminals, with different parameters, a number of terminals that changes with time, different classes of tasks with priorities, etc. SSJ offers the tools to implement these generalizations easily. The program would be more elaborate but its structure would be similar.

```
import java.io.*;

public class TimeShared {

   int nbTerminal = 20;     // Number of terminals.
   double quantum;          // Quantum size.
   double overhead = 0.001; // Amount of overhead (h).
   double meanThink = 5.0;  // Mean thinking time.
   double alpha  = 0.5;
   double lambda = 1.0;     // Parameters of the Weibull service needs.
   int N     = 1100;        // Total number of tasks to simulate.
   int N0    = 100;         // Number of tasks for warmup.
   int nbRep = 10;          // Number of simulation runs (R).
   int nbTasks;             // Number of tasks ended so far.

   double q1 = 0.1;         // Quantum size for first config.
   double q2 = 0.2;         // Quantum size for second config.

   RandomStream genThink = new RandMrg ("Gen. for thinking times");
   RandomStream genServ  = new RandMrg ("Gen. for service requirements");
   Resource server       = new Resource (1, "The server");
   Tally meanInRep       = new Tally ("Average for current run");
   Tally statDiff        = new Tally ("Differences on mean response times");

   double meanConf1[];      // To store the results for first config.

   public static void main (String[] args) { new TimeShared (); }

   public TimeShared () {
      meanConf1 = new double[nbRep];
      for (int conf = 1; conf <= 2; conf++) {
         if (conf == 1) quantum = q1;
         else {            // Start second configuration: reset seeds.
            quantum = q2;
            genThink.resetStartStream ();
            genServ.resetStartStream ();
            }
         for (int rep = 0; rep < nbRep; rep++) {
            simulOneRun ();
            if (conf==1) meanConf1[rep] = meanInRep.average();
            else statDiff.update (meanConf1[rep] - meanInRep.average());
            genThink.resetNextSubstream ();
            genServ.resetNextSubstream ();
            }
         }
      statDiff.printConfIntStudent (0.9);
      }

   private void simulOneRun () {
      Sim.init();
      server.init();
      meanInRep.init();
      nbTasks = 0;
      for (int i=1; i <= nbTerminal; i++) new Terminal().schedule (0.0);
      Sim.start();
      }
```

Figure 8: Simulation of a time shared system.

```
class Terminal extends Process {
    public void actions () {
       double arrivTime;    // Arrival time of current request.
       double timeNeeded;   // Server's time still needed for it.
       while (nbTasks < N) {
          delay (Rand1.expon (genThink, meanThink));
          arrivTime = Sim.time ();
          timeNeeded = Rand1.weibull (genServ, alpha, lambda);
          while (timeNeeded > quantum) {
             server.request (1);
             delay (quantum + overhead);
             timeNeeded -= quantum;
             server.release (1);
             }
          server.request(1);  // Here, timeNeeded <= quantum.
          delay (timeNeeded + overhead);
          server.release(1);
          nbTasks++;
          if (nbTasks > N0) meanInRep.update (Sim.time() - arrivTime);
                            // Take the observation if warmup is over.
          }
       Sim.stop();           // N tasks have now completed.
       }
    }
}
```

Figure 8: Simulation of a time shared system (continued).

```
REPORT on Tally stat. collector ==> Differences on mean response times
   min          max        average      standard dev.   nb. obs.
   -0,134      0,369        0,168         0,175            10

        90.0 confidence interval for mean ( 0,067, 0,269 )
```

Figure 9: Difference in the mean response times for $q = 0.1$ and $q = 0.2$ for the time shared system.

3.5 A simplified bank

This is Example 1.4.1 of ifnextchar [tempswatruecitexptempswafalsecitexp[]sBRA87a, page 14. A bank has a random number of tellers every morning. On any given day, the bank has t tellers with probability q_t, where $q_3 = 0.80$, $q_2 = 0.15$, and $q_1 = 0.05$. All the tellers are assumed to be identical from the modeling viewpoint.

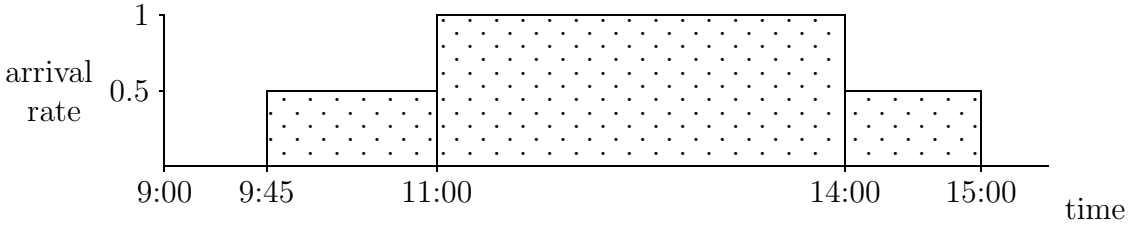


Figure 10: Arrival rate of customers to the bank.

The bank opens at 10:00 and closes at 15:00 (i.e., 3 P.M.). The customers arrive randomly according to a Poisson process with piecewise constant rate $\lambda(t)$, $t \geq 0$. The arrival rate $\lambda(t)$ (see Fig. 10) is 0.5 customer per minute from 9:45 until 11:00 and from 14:00 until 15:00, and one customer per minute from 11:00 until 14:00. The customers who arrive before 9:45 join a FIFO queue and wait for the bank to open. At 15:00, the door is closed, but all the customers already in will be served. Service starts at 10:00.

Customers form a FIFO queue for the tellers, with balking. An arriving customer will balk (walk out) with probability p_k if there are k customers ahead of him in the queue (not counting the people receiving service), where

$$p_k = \begin{cases} 0 & \text{if } k \leq 5; \\ (n-5)/5 & \text{if } 5 < k < 10; \\ 1 & \text{if } k \geq 10. \end{cases}$$

The customer service times are independent Erlang random variables: Each service time is the sum of two independent exponential random variables with mean one.

We want to estimate the expected number of customers served in a day, and the expected average wait for the customers served on a day.

Figures 11 and 12 give simulation programs for this bank model. The first program uses only events and the second one views the customers as processes. Both programs have events at the fixed times 9:45, 10:00, etc.; these events are implicit in the process-oriented implementation. At 9:45, the counters are initialized and the arrival process is started. The time until the first arrival, or the time between one arrival and the next one, is (tentatively) an exponential with a mean of 2 minutes. However, as soon as an arrival turns out to be past 11:00, its time must be readjusted to take into account the increase of the arrival rate at 11:00. The event 11:00 takes care of this readjustment, and the event at 14:00 makes a similar readjustment when the arrival rate decreases. We give the specific name `nextArriv` to the next planned arrival event in the event-oriented case, and `nextCust` to the next customer scheduled to arrive in the process-oriented case, in order to be able to reschedule

```
public class BankEv {

   double   minute = 1.0 / 60.0;
   long     nbTellers;            // Number of tellers.
   int      nbBusy;               // Number of tellers busy.
   int      nbWait;               // Queue length.
   int      nbServed;             // Number of customers served so far.
   double   meanDelay;            // Mean time between arrivals.
   Event    nextArriv  = new Arrival ();  // The next arrival.
   RandMrg  genArr     = new RandMrg ();  // For customer's arrivals.
   RandMrg  genServ    = new RandMrg ();  // For service times.
   RandMrg  genTeller  = new RandMrg ();  // For number of tellers.
   RandMrg  genBalk    = new RandMrg ();  // For balking decisions.
   Tally    statServed = new Tally ("Nb. served per day");
   Tally    avWait     = new Tally ("Average wait per day (hours)");
   Accumulate wait     = new Accumulate ("cumulated wait for this day");

   public static void main (String[] args) { new BankEv(); }

   public BankEv() {
      for (int i=1; i<=100; i++)  simulOneDay();
      statServed.report();   avWait.report();
      }

   public void simulOneDay () {
      Sim.init();       wait.init();
      nbTellers = 0;    nbBusy    = 0;
      nbWait    = 0;    nbServed  = 0;
      e9h45.schedule (9.75);
      e10h.schedule (10.0);
      e11h.schedule (11.0);
      e14h.schedule (14.0);
      e15h.schedule (15.0);
      Sim.start();
      statServed.update (nbServed);
      avWait.update (wait.sum());
      }

   Event e9h45 = new Event() {
      public void actions() {
         meanDelay = 2.0 * minute;
         nextArriv.schedule (Rand1.expon (genArr, meanDelay));
         }
      };

   Event e10h = new Event() {
      public void actions() {
         double u = genTeller.randU01();
         if (u >= 0.2) nbTellers = 3;
         else if (u < 0.05) nbTellers = 1;
         else nbTellers = 2;
         while (nbWait > 0 && nbBusy < nbTellers) {
         nbBusy++;  nbWait--;
            new Departure().schedule (Rand1.erlang (genServ, 2, minute));
            }
         wait.update (nbWait);
         }
      };
```

Figure 11: Event-oriented simulation of the bank model.

```
Event e11h = new Event() {
   public void actions() {
      nextArriv.reschedule ((nextArriv.time() - Sim.time()) / 2.0);
      meanDelay = minute;
      }
   };

Event e14h = new Event() {
   public void actions() {
      nextArriv.reschedule ((nextArriv.time() - Sim.time()) * 2.0);
      meanDelay = 2.0 * minute;
      }
   };

Event e15h = new Event() {
   public void actions() { nextArriv.cancel(); }
   };

private boolean balk() {
   return ((nbWait>9) ||
          ((nbWait>5) && (5.0 * genBalk.randU01() < nbWait-5)));
   }

class Arrival extends Event {
   public void actions() {
      nextArriv.schedule (Rand1.expon (genArr, meanDelay));
      if (nbBusy < nbTellers) {
         nbBusy++;
         new Departure().schedule (Rand1.erlang (genServ, 2, minute));
         }
      else if (!balk())  { nbWait++;  wait.update (nbWait); }
      }
  }

class Departure extends  Event {
   public void actions() {
      nbServed++;
      if (nbWait>0) {
         new Departure().schedule (Rand1.erlang (genServ, 2, minute));
         nbWait--;   wait.update (nbWait);
         }
      else nbBusy--;
      }
   };
}
```

Figure 11: Event-oriented simulation of the bank model (continuation).

```
public class BankProc {

   double   minute = 1.0 / 60.0;
   int      nbServed;             // Number of customers served so far.
   double   meanDelay;            // Mean time between arrivals.
   Process  nextCust;             // Next customer to arrive.
   Resource tellers    = new Resource (0, "The tellers");
   RandMrg  genArr     = new RandMrg ();  // For customer's arrivals.
   RandMrg  genServ    = new RandMrg ();  // For service times.
   RandMrg  genTeller  = new RandMrg ();  // For number of tellers.
   RandMrg  genBalk    = new RandMrg ();  // For balking decisions.
   Tally    statServed = new Tally ("Nb. served per day");
   Tally    avWait     = new Tally ("Average wait per day (hours)");

   public static void main (String[] args) { new BankProc(); }

   public BankProc() {
      tellers.waitList().collectStat (true);
      for (int i=1; i<=100; i++)  simulOneDay();
      statServed.report();   avWait.report();
      }

   public void simulOneDay () {
      Sim.init();
      new OneDay().schedule (9.75);
      Sim.start ();
      statServed.update (nbServed);
      avWait.update (tellers.waitList().statSize().sum());
      }

   class OneDay extends Process {
      public void actions () {
         int nbTellers;         // Number of tellers today.
         nbServed = 0;
         tellers.setCapacity (0);
         tellers.waitList().initStat();
         meanDelay = 2.0 * minute;
         // It is 9:45, start arrival process.
         (nextCust = new Customer ()).schedule (Rand1.expon (genArr, meanDelay));
         delay (15.0 * minute);
         // Bank opens at 10:00, generate number of tellers.
         double u = genTeller.randU01();
         if (u >= 0.2) nbTellers = 3;
         else if (u < 0.05)  nbTellers = 1;
         else nbTellers = 2;
         tellers.setCapacity (nbTellers);
         delay (1.0);   // It is 11:00, double arrival rate.
         nextCust.reschedule (nextCust.getDelay() / 2.0);
         meanDelay = minute;
         delay (3.0);   // It is 14:00, halve arrival rate.
         nextCust.reschedule (nextCust.getDelay() * 2.0);
         meanDelay = 2.0 * minute;
         delay (1.0);   // It is 15:00, bank closes.
         nextCust.cancel();
         }
      }
```

Figure 12: Process-oriented simulation of the bank model.

```
class Customer extends Process {
   public void actions () {
      (nextCust = new Customer()).schedule (Rand1.expon (genArr, meanDelay));
      if (!balk()) {
         tellers.request (1);
         delay (Rand1.erlang (genServ, 2, minute));
         tellers.release (1);
         nbServed++;
         }
      }

   private boolean balk() {
      int n = tellers.waitList().size();
      return ((n>9) || ((n>5) && (5.0 * genBalk.randU01() < n-5)));
      }
   }
}
```

Figure 12: Process-oriented simulation of the bank model (continuation).

that particular event (or process) to a different time. Note that in the event-oriented case, a *single* arrival event is created at the beginning and this same event is scheduled over and over again. This can be done because there is never more than one arrival event in the event list. We cannot do this for the customer processes in the process-oriented case, however, because several processes can be alive simultaneously.

At the bank opening at 10:00, an event generates the number of tellers and starts the service for the corresponding customers. The event at 15:00 cancels the next arrival.

Upon arrival, a customer checks if a teller is free. If so, one teller becomes busy and the customer generates its service time and schedules his departure, otherwise the customer either balks or joins the queue. The balking decision is computed by the method `balk`, using the random number stream `genBalk`. The arrival event also generates the next scheduled arrival. Upon departure, the customer frees the teller, and the first customer in the queue, if any, can start its service.

The constructor (`BankEv` or `Bankproc`) simulates the bank for 100 days and prints a statistical report. If X_i is the number of customers served on day i and Q_i the total waiting time on day i, the program estimates $E[X_i]$ and $E[Q_i]$ by their sample averages $\bar{X}_n$ and $\bar{Q}_n$ with $n = 100$. For each simulation run (each day), `simulOneDay` initializes the clock, event list, and statistical probe for the waiting times, schedules the deterministic events, and runs the simulation. After 15:00, no more arrival occurs and the event list becomes empty when the last customer departs. At that point, the program returns to right after the `Sim.start()` statement and updates the statistical counters for the number of customers served during the day and their total waiting time.

The process-oriented version of the program is shorter, because certain aspects (such as the details of an arrival or departure event) are taken care of automatically by the process/resource construct, and the events 9:45, 10:00, etc., are replaced by a single process.

At 10 o'clock, the `setCapacity` statement that fixes the number of tellers also takes care of starting service for the appropriate number of customers.

These two programs give the same results, shown in Figure 13. However, the process-oriented program take approximately 4 to 5 times longer to run than its event-oriented counterpart.

```
REPORT on Tally stat. collector ==> Nb. served per day
   min         max         average      standard dev.   nb. obs.
   152         285         240,59         19,21           100

REPORT on Tally stat. collector ==> Average wait per day (hours)
   min         max         average      standard dev.   nb. obs.
  0,816      35,613        4,793         5,186           100
```

Figure 13: Results of the `BankEv` and `BankProc` programs.

3.6 Guided visits

This example is translated from [11]. A touristic attraction offers guided visits, using three guides. The site opens at 10:00 and closes at 16:00. Visitors arrive in small groups (e.g., families) and the arrival process of those groups is assumed to be a Poisson process with rate of 20 groups per hour, from 9:45 until 16:00. The visitors arriving before 10:00 must wait for the opening. After 16:00, the visits already under way can be completed, but no new visit is undertaken, so that all the visitors still waiting cannot visit the site and are lost.

The size of each arriving group of visitors is a discrete random variable taking the value i with probability p_i given in the following table:

i	1	2	3	4
p_i	.2	.6	.1	.1

Visits are normally made by groups of 8 to 15 visitors. Each visit requires one guide and lasts 45 minutes. People waiting for guides form a single queue. When a guide becomes free, if there is less than 8 people in the queue, the guide waits until the queue grows to at least 8 people, otherwise she starts a new visit right away. If the queue contains more than 15 people, the first 15 will go on this visit. At 16:00, if there is less than 8 people in the queue and a guide is free, she starts a visit with the remaining people. At noon, each free guide takes 30 minutes for lunch. The guides that are busy at that time will take 30 minutes for lunch as soon as they complete their on-going visit.

Sometimes, an arriving group of visitors may decide to just go away (balk) because the queue is too long. We assume that the probability of balking when the queue size is n is given by

$$R(n) = \begin{cases} 0 & \text{for } n \leq 10; \\ (n-10)/30 & \text{for } 10 < n < 40; \\ 1 & \text{for } n \geq 40. \end{cases}$$

The aim is to estimate the average number of visitors lost per day, in the long run. The visitors lost are those that balk or are still in the queue after 16:00.

A simulation program for this model is given in Figure 14. Here, time is measured in hours, starting at midnight. At time 9:45, for example, the simulation clock is at 9.75. The (process) class `Guide` describes the daily behavior of a guide (each guide is an instance of this class), whereas `Arrival` generates the arrivals according to a Poisson process, the group sizes, and the balking decisions. The event `closing` closes the site at 16:00.

The `Bin` mechanism `visitReady` is used to synchronize the `Guide` processes. The number of tokens in this bin is 1 if there is enough visitors in the queue to start a visit (8 or more) and is 0 otherwise. When the queue size reaches 8 due to a new arrival, the `Arrival` process puts a token into the bin. This wakes up a guide if one is free. A guide must take a token from the bin to start a new visit. If there is still 8 people or more in the queue when she starts the visit, she puts the token back to the bin immediately, to indicate that another visit is ready to be undertaken by the next available guide.

The simulation results are in Figure 15.

```
class Visits {

   int queueSize;    // Size of waiting queue.
   int nbLost;       // Number of visitors lost so far today.
   Bin visitReady = new Bin ("Visit ready");
      // A token becomes available when there is enough visitors
      // to start a new visit.
   Tally avLost = new Tally ("Nb. of visitors lost per day");
   RandMrg  genArriv = new RandMrg();  // For times between arrivals.
   RandMrg  genSize  = new RandMrg();  // For sizes of arriving group.
   RandMrg  genBalk  = new RandMrg();  // For balking decisions.

   static public void main (String[] args) { new Visits(); }

   public Visits() {
      for (int i = 1; i <= 100; i++) oneDay();
      avLost.printConfIntStudent (0.9);
      }

   private void oneDay() {
      queueSize = 0;   nbLost = 0;
      Sim.init();
      visitReady.init();
      closing.schedule (16.0);
      new Arrival().schedule (9.75);
      for (int i=1; i<=3; i++) new Guide ().schedule (10.0);
      Sim.start ();
      avLost.update (nbLost);
      }

   Event closing = new Event() {
      public void actions() {
         if (visitReady.waitList().size() == 0) nbLost += queueSize;
         Sim.stop();
         }
      };

   class Guide extends Process {
      public void actions() {
         boolean lunchDone = false;
         while (true) {
            if (Sim.time() > 12.0 && !lunchDone) {
               delay (0.5);  lunchDone = true;
               }
            visitReady.take (1);  // Starts the next visit.
            if (queueSize > 15) queueSize -= 15;
            else queueSize = 0;
            if (queueSize >= 8) visitReady.put (1);
                                  // Enough people for another visit.
            delay (0.75);
            }
         }
      }
```

Figure 14: Guided visits

```
class Arrival extends Process {
   public void actions() {
      while (true) {
         delay (Rand1.expon (genArriv, 0.05));
         // A new group of visitors arrives.
         int groupSize;  // number of visitors in group.
         double u = genSize.randU01();
         if (u <= 0.2)      groupSize = 1;
         else if (u <= 0.8) groupSize = 2;
         else if (u <= 0.9) groupSize = 3;
         else               groupSize = 4;
         if (!balk()) {
            queueSize += groupSize;
            if (queueSize >= 8 && visitReady.getAvailable() == 0)
               visitReady.put (1);  // A token is now available.
            }
         else  nbLost += groupSize;
         }
      }

   private boolean balk() {
      if (queueSize <= 10)  return false;
      if (queueSize >= 40)  return true;
      return (genBalk.randU01() < ((queueSize - 10.0) / 30.0));
      }
   }
}
```

Figure 14: Guided visits (continued)

```
REPORT on Tally stat. collector ==> Nb. of visitors lost per day
   min        max        average      standard dev.   nb. obs.
    3         48          21.78         10.639          100

        90.0 confidence interval for mean ( 20.014, 23.546 )
```

Figure 15: Simulation results for the guided visits model.

Could we have used a `Condition` instead of a `Bin` for synchronizing the `Guide` processes? The problem would be that if several guides are waiting for a condition indicating that the queue size has reached 8, *all* these guides (not only the first one) would resume their execution simultaneously when the condition becomes true.

APPENDIX A.

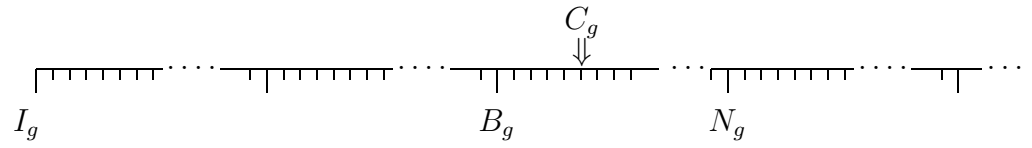
The SSJ Classes

RandomStream

This interface defines the basic structures to handle multiple streams of uniform (pseudo)-random numbers and convenient tools to move around within and across these streams. The actual random number generators (RNGs) are provided in classes that implement this `RandomStream` interface. A variety of such RNGs are available, in classes `RandMrg`, `RandTaus`, etc. Each stream of random numbers is an object of the class that implements this interface, and can be viewed as a virtual random number generator.

For each type of base RNG (i.e., each implementation of the `RandomStream` interface, the full period of the generator is cut into adjacent *streams* (or segments) of length Z, and each of these streams is partitioned into V *substreams* of length W, where $Z = VW$. The values of V and W depend on the specific RNG, but are usually larger than 2^{50}. Thus, the distance Z between the starting points of the successive streams provided by an RNG usually exceeds 2^{100}. The initial seed of the RNG is the starting point of the first stream. It has a default value for each type of RNG, but this initial value can be changed by calling `setPackageSeed` for the corresponding class. Each time a new `RandomStream` is created, its starting point (initial seed) is computed automatically, Z steps ahead of the starting point of the previously created stream of the same type, and its current state is set equal to this starting point.

For each stream, one can advance by one step and generate one value, or go ahead to the beginning of the next substream within this stream, or go back to the beginning of the current substream, or to the beginning of the stream, or jump ahead or back by an arbitrary number of steps. Denote by C_g the current state of a stream g, I_g its initial state, B_g the state at the beginning of the current substream, and N_g the state at the beginning of the next substream. The following diagram shows an example of a stream whose state is at the 6th value of the third substream, i.e., $2W + 5$ steps ahead of its initial state I_g and 5 steps ahead of its state B_g. The form of the state of a stream depends on its type. For exemple, the state of a stream of class `RandMrg` is a vector of 6 integers represented internally in floating point (in an array of `double`).



The methods for manipulating the streams and generating random numbers are implemented differently for each type of RNG. The methods whose formal parameter types do not depend on the RNG type are specified in the interface `RandomStream`. The others (e.g., for setting the seeds) are given only in the classes that implement the specific RNG types.

See [9, 13, 15] and Section 3.4 for examples of situations where the multiple streams offered here are useful.

Methods for generating random variates from non-uniform distributions are provided in classes `Rand1`, `Rand2`, etc. The class `DiscreteDist` can be used to construct an arbitrary discrete distribution over a finite set of values, and to generate from it.

`public interface RandomStream`

Methods

`public void resetStartStream ();`

Reinitializes the stream to its initial state I_g: C_g and B_g are set to I_g.

`public void resetStartSubstream ();`

Reinitializes the stream to the beginning of its current substream: C_g is set to B_g.

`public void resetNextSubstream ();`

Reinitializes the stream to the beginning of its next substream: N_g is computed, and C_g and B_g are set to N_g.

`public void setAntithetic (boolean a);`

After this method is called with `a = true`, the stream starts generating antithetic variates, i.e., $1 - U$ instead of U, until the method is called again with `a = false`.

`public void writeState ();`

Prints (to standard output) the current state of this stream.

`public double randU01 ();`

Returns a (pseudo)random number from the uniform distribution over the interval $(0, 1)$, using this stream, after advancing its state by one step.

`public int randInt (int i, int j);`

Returns a (pseudo)random number from the discrete uniform distribution over the integers $\{i, i + 1, \ldots, j\}$, using this stream. (Calls `randU01` once.)

RandMrg

This class implements the interface `RandomStream`, with a few additional tools. The backbone (or main) generator is the combined multiple recursive generator (CMRG) `Mrg32k3a` proposed by L'Ecuyer [12], implemented in 64-bit floating-point arithmetic. This backbone generator has period length $\rho \approx 2^{191}$. The values of V, W, and Z are 2^{51}, 2^{76}, and 2^{127}, respectively. (See `RandomStream` for their definition.) The seed of the RNG, and the state of a stream at any given step, are 6-dimensional vectors of 32-bit integers. The default initial seed of the RNG is $(12345, 12345, 12345, 12345, 12345, 12345)$.

```
public class RandMrg implements RandomStream
```

Constructors

`public RandMrg()`

Constructs a new stream, initializes its seed I_g, sets B_g and C_g equal to I_g, and sets its antithetic switch to `false`. The seed I_g is equal to the initial seed of the package given by `setPackageSeed` if this is the first stream created, otherwise it is Z steps ahead of that of the stream most recently created in this class.

`public RandMrg (String name)`

Constructs a new stream with an identifier `name` (can be used when printing the stream state, in error messages, etc.).

Methods

`public static void setPackageSeed (long seed[])`

Sets the initial seed for the class `RandMrg` to the six integers in the vector `seed[0..5]`. This will be the seed (initial state) of the first stream. If this procedure is not called, the default initial seed is $(12345, 12345, 12345, 12345, 12345, 12345)$. If it is called, the first 3 values of the seed must all be less than $m_1 = 4294967087$, and not all 0; and the last 3 values must all be less than $m_2 = 4294944443$, and not all 0.

`public void resetStartStream ()`

`public void resetStartSubstream ()`

`public void resetNextSubstream ()`

See `RandomStream`.

`public void increasedPrecis (boolean incp)`

After calling this procedure with `incp = true`, each call to the generator (direct or indirect) for this stream will return a uniform random number with (roughly) 53 bits of resolution instead of 32 bits, and will advance the state of the stream by 2 steps instead of 1. More precisely, if `s` is a stream of the class `RandMrg`, in the non-antithetic case, the instruction "`u = s.randU01()`" when the resolution has been increased is equivalent to "`u = (s.randU01()`

`+ s.randU01(g) * fact) % 1.0`" where the constant `fact` is equal to 2^{-24}. This also applies when calling `randU01` indirectly (e.g., via `randInt`, etc.).

By default, or if this procedure is called again with `incp = false`, each call to `randU01` for this stream advances the state by 1 step and returns a number with 32 bits of resolution.

`public void setAntithetic (boolean a)`

After this procedure is called with `a = true`, the stream starts generating antithetic variates, i.e., $1 - U$ instead of U, until the procedure is called again with `a = false`.

`public void advanceState (int e, int c)`

Advances the state of this stream by k values, without modifying the states of other streams (as in `setSeed`), nor the values of B_g and I_g associated with this stream. If $e > 0$, then $k = 2^e + c$; if $e < 0$, then $k = -2^{-e} + c$; and if $e = 0$, then $k = c$. Note: c is allowed to take negative values. This method should be used only in very exceptional cases; proper use of the `reset...` methods and of the stream constructor cover most reasonable situations.

`public void setSeed (long seed[])`

Sets the initial seed I_g of this stream to the vector `seed[0..5]`. This vector must satisfy the same conditions as in `setPackageSeed`. The stream is then reset to this initial seed. The states and seeds of the other streams are not modified. As a result, after calling this procedure, the initial seeds of the streams are no longer spaced Z values apart. For this reason, this method should be used only in very exceptional situations; proper use of `reset...` and of the stream constructor is preferable.

`public double[] getState()`

Returns the current state C_g of this stream. This is a vector of 6 integers represented in floating point. This method is convenient if we want to save the state for subsequent use.

`public void writeState ()`

Prints (to standard output) the name and current state C_g of this stream.

`public void writeStateFull ()`

Prints (to standard output) the name of this stream and the values of all its internal variables.

`public double randU01 ()`

Returns a (pseudo)random number from the uniform distribution over the interval $(0, 1)$, using this stream, after advancing its state by one step. Normally, the returned number has 32 bits of resolution, in the sense that it is always a multiple of $1/(2^{32} - 208)$. However, if the precision has been increased by calling `increasedPrecis` for this stream, the resolution is higher and the stream state advances by two steps.

`public int randInt (int i, int j)`

Returns a (pseudo)random number from the discrete uniform distribution over the integers $\{i, i + 1, \ldots, j\}$, using this stream. Makes one call to `randU01`.

Rand1

This static class provides a few methods for generating random variates from various distributions, using a given random stream. All the methods given here use inversion [4, 9].

```
public class Rand1
```

Methods

`public static double uniform (RandomStream s, double a, double b)`

Returns a random variate having the uniform distribution over the real interval (a, b), using stream `s`. The returned value is always *strictly* between a and b.

`public static double expon (RandomStream s, double mean)`

Returns a random variate having the exponential distribution with mean $\mu =$ `mean`, i.e., with distribution function $F(x) = 1 - e^{-x/\mu}$ for $x > 0$, using stream `s`. Uses inversion.

`public static double erlang (RandomStream s, int k, double mu)`

Returns a random variate having the Erlang distribution with parameters (k, μ), which is the sum of k exponential random variables with mean μ. The mean of this Erlang random variable is $k\mu$. This method simply generates k exponentials by calling `expon` k times with stream `s`, and returns the sum.

```
public static double weibull (RandomStream s,
                              double alpha, double lambda)
```

Returns a random variate having the Weibull distribution with parameters $\alpha =$ `alpha` and $\lambda =$ `lambda`, using stream `s`. This Weibull distribution function is $F(x) = 1 - \exp(-\lambda x^\alpha)$ for $x > 0$. The mean is given by $\Gamma(1 + 1/\alpha)/\lambda^{1/\alpha}$. Restrictions: must have $\alpha > 0$ et $\lambda > 0$. Uses inversion.

`public static double normalDist (double x)`

Returns $y = F(x)$, where F is the standard normal distribution function, with mean 0 and variance 1. Uses a rational approximation giving an absolute error less than 10^{-15} over the full range (see [6], page 90).

`public static double invNormalDist (double u)`

Returns $y = F^{-1}(u)$, where F is the standard normal distribution function, with mean 0 and variance 1. Uses a rational approximation giving at least 7 decimal digits of precision for $10^{-20} < u < 1 - 10^{-20}$ (see [6], page 85).

`public static double normal (RandomStream s, double mean, double sdev)`

Returns a normal random variate with mean $\mu =$ `mean` and standard deviation $\sigma =$ `sdev`, using stream `s`. Uses inversion via `invNormalDist`.

`public static double invStudentDist (int n, double u)`

Returns $y = F^{-1}(u)$, where F is the Student distribution function with `n` degrees of freedom.

`public static double student (RandomStream s, int n)`

Returns a random variate having the Student distribution with `n` degrees of freedom, using stream `s`. Uses inversion via `invStudentDist`.

StatProbe

The objects of this class are *statistical probes* or *collectors*, which are elementary devices for collecting statistics. Each probe collects statistics on a given variable. The subclasses `Tally` and `Accumulate` implement two specific types of probes, for the case of successive observations $X_1, X_2, X_3, \ldots$, and for the case of a variable whose value evolves in time, respectively.

```
public abstract class StatProbe
```

Methods

`abstract public void init ();`

Initializes the statistical collector.

`abstract public void update (double x);`

Gives a new observation `x` to the statistical collector.

`public double min ()`

Returns the smallest value taken by the variable (i.e., passed by the `update` method) since the last initialization of this probe.

`public double max ()`

Returns the largest value taken by the variable (i.e., passed by the `update` method) since the last initialization of this probe.

`public double sum ()`

Returns the sum cumulated so far for this probe. The meaning of this sum depends on the subclass (`Tally` or `Accumulate`).

`abstract public double average ();`

Returns the average for this collector.

`abstract public void report ();`

Prints a report for this statistical collector.

Tally

A subclass of `StatProbe`. This type of statistical collector takes a sequence of real-valued observations $X_1, X_2, X_3, \ldots$ and can return the average, the variance, a confidence interval for the theoretical mean, etc. Each `update` call provides a new observation. The collector can be reinitialized by `init`.

```
public class Tally extends StatProbe
```

Constructors

`public Tally ()`
Construct a new `Tally` statistical probe.

`public Tally (String name)`
Construct a new `Tally` statistical probe with descriptor `name`.

Methods

`public void init ()`
Initializes the statistical collector.

`public void update (double x)`
Gives a new observation `x` to the statistical collector.

`public double min ()`
Returns the smallest observation so far.

`public double max ()`
Returns the largest observation so far.

`public double sum ()`
Returns the sum of all observations.

`public int numberObs ()`
Returns the number of observations given to this probe since its last initialization.

`public double average ()`
Returns the average value of the observations since the last initialization.

`public double variance()`
Returns the variance of the observations since the last initialization.

`public double standardDev()`
Returns the standard deviation of the observations since the last initialization.

`public void confIntStudent (double level, double[] centerAndRadius)`

Returns the center and half-length (radius) of a confidence interval of confidence level `level` for the true mean of the random variable X, assuming that the observations given to this collector are independent observations of X. The center and radius are returned in elements 0 and 1 of the array object `centerAndRadius[]`. This interval is computed based on the statistic

$$T = \frac{\overline{X} - \mu}{S_x / \sqrt{n}}$$

where n is the number of observations given to this collector since its last initialization, $\overline{X}$ = `average()` is the average of these observations, S_x = `standardDev()` is the empirical standard deviation. Under the assumption that the observations of X are independent and identically distributed, with the normal distribution, T has the Student distribution with $n - 1$ degrees of freedom. The confidence interval given by this method is valid *only if* this assumption is approximately verified, or if n is large enough so that $\overline{X}$ is approximately normally distributed.

`public void printConfIntStudent (double level)`

Same as `confIntStudent`, but prints the confidence interval instead of returning it.

`public void report ()`

Prints a report on this collector since its last initialization.

Accumulate

A subclass of `StatProbe`, for collecting statistics on a variable that evolves in time, with a piecewise-constant trajectory. Each time the variable changes its value, the method `update` must be called to inform the probe of the new value. The probe can be reinitialized by `init`.

```
public class Accumulate extends StatProbe
```

Constructors

`public Accumulate ()`

Construct and initializes a new `Accumulate` statistical probe.

`public Accumulate (String name)`

Construct and initializes a new `Accumulate` statistical probe with descriptor `name`.

Methods

`public void init ()`

Initializes the statistical collector and puts the current value of the corresponding variable to 0. A call to `init` should normally be followed immediately by a call to `update` to give the value of the variable at the initialization time.

`public void init (double x)`

Same as `init` followed by `update(x)`.

`public void update (double x)`

Gives a new observation `x` to the statistical collector.

`public double min ()`

Returns the smallest value taken by the variable since the last initialization.

`public double max ()`

Returns the largest value taken by the variable since the last initialization.

`public double sum ()`

Returns the sum cumulated so far for this probe.

`public double average ()`

Returns the time-average since the last initialization.

`public void report ()`

Prints a report on this collector since its last initialization.

List

Objects of this class are lists that can contain any type of object. These lists are implemented as doubly linked lists. Methods are available to insert objects in a list, remove objects from a list, view objects at certain positions in a list, obtain statistics on the size of the list and the sojourn times of objects in it, and so on.

Each list has a pointer indicating the last place where an operation has been performed (except in `remove`). This place is called the *current position*. One can use this pointer, for example, to scan the list by calling `view (FIRST)` followed by a series of calls to `view (NEXT)`.

```
public class List
```

```
public static final int FIRST
public static final int LAST
public static final int PREVIOUS
public static final int NEXT
public static final int CURRENT
```

Place where a object can be observed, removed, or inserted in a list. `FIRST` means at the beginning, `LAST` means at the end, `PREVIOUS` means just before the current position, `NEXT` means just after the current position, `CURRENT` means at the current position (and cannot be used for insertion).

Constructors

```
public List()
```

Constructs a new list, initially empty.

```
public List (String name)
```

Constructs a new list with descriptor `name`. This descriptor can be used to identify the list in traces and reports.

Methods

```
public void collectStat (boolean flag)
```

If `flag = true`, this method starts collecting statistics on this list. It creates two statistical probes. The first one, of the class `Accumulate`, measures the evolution of the size of the list as a function of time. It can be accessed by the method `statSize`. The second one, of the class `Tally`, and accessible via `statSojourn`, samples the sojourn times in the list of the objects removed during the observation period, i.e., between the last initialization time of this statistical probe and the current time. The method `collectStat` automatically calls `initStat` to initialize these two probes. When this method is used, it is normally invoked immediately after calling the constructor of the list. If `flag = false`, this method stops the collecting of statistics.

`public void initStat()`

Reinitializes the two statistical probes created by `collectStat` and makes an update for the probe on the list size.

`public void init()`

Empties this list and initializes its statistical probes if `collectStat` has been called.

`public void insert (Object obj, int where)`

Inserts `obj` in the list, at the position specified by `where`. The position of this newly inserted object becomes the current position.

`public Object view (int where)`

Returns the object at position `where` in the list, without removing it. The position of this object becomes the current position. If this object does not exist (e.g., if we ask for `PREVIOUS` while the current position is at the beginning of the list, or if the list is empty), returns `null` and the current position remains unchanged.

`public boolean belongs (Object obj)`

If `obj` is in this list, returns `true` and the current position now points to this object; otherwise, returns `false` and the current position is unchanged.

`public Object remove (int where)`

Returns and removes from the list the object at the position indicated by `where`. Returns `null` if this object does not exist (e.g., if we ask for `PREVIOUS` while the current position is at the beginning of the list, or if the list is empty). If the element to be removed is at the current position and the list is not becoming empty, then the current position will point to the object following the removed one (or to the end of the list if the removed object was at the end of the list).

`public int size ()`

Returns the number of objects currently in the list.

`public Accumulate statSize ()`

Returns the statistical probe on the evolution of the size of the list as a function of the simulation time. This probe exists only if `collectStat` has been called for this list.

`public Tally statSojourn ()`

Returns the statistical probe on the sojourn times of the objects in the list. This probe exists only if `collectStat` has been called for this list.

`public void report ()`

Prints a statistical report on the list, provided that `collectStat` has been called before for this list.

Sim

This static class contains the executive of a discrete-event simulation, which maintains the simulation clock and starts executing the events and processes in the appropriate order. Its methods permit one to start, stop, and (re)initialize the simulation, and read the simulation clock.

```
public abstract class Sim implements Runnable
```

Methods

`public static double time ()`

Returns the current value of the simulation clock.

`public static void init ()`

Reinitializes the simulation executive by clearing up the event list, killing all the `Process` objects (if any), and resetting the simulation clock to zero.

`public static void init (EventList evlist)`

Same as `init`, but also chooses `evlist` as the event list to be used. For example, `init (new DoublyLinked)` initializes the simulation with a doubly linked linear structure for the event list.

`public static void start ()`

Starts the simulation executive. There must be at least one `Event` in the event list when this method is called.

`public static void stop ()`

Tells the simulation executive to stop as soon as it takes control, and to return control to the program that called `Sim.start`. This program will then continue executing from the instructions right after its call to `Sim.start`. If an `Event` is currently executing (and this event has just called `Sim.stop`), the executive will take control when the event terminates its execution. If a `Process` is currently executing, the executive will take control when the process terminates or suspends itself by invoking one of its methods such as `delay`, `suspend`, `request`, etc.

Event

This abstract class provides event scheduling tools. Each type of event should be defined as a subclass of the class `Event`, and should provide an implementation of the method `actions` which is executed when an event of this type occurs. The instances of these subclasses are the actual events.

An event can be created and scheduled simultaneously by the constructor, or it can be created first and then scheduled separately by calling one of the scheduling methods `schedule`, `scheduleNext`, `scheduleBefore`, etc. An event can also be cancelled before it occurs.

```
public abstract class Event
```

```
   public static String descriptor;
```

This string can be reinitialized (optional) within each subclass of `Event`, to describe the subclass (e.g., for error messages, traces, etc.). By default (and for the general `Event` class), this is the empty string.

Constructors

```
   public Event ()
```

Constructs a new event instance, which can be placed afterwards into the event list by calling one of the `schedule...` variants. For example, if `Bang` is an `Event` subclass, the statement "`new Bang().scheduleNext();`" creates a new `Bang` event and places it at the beginning of the event list.

```
   public Event (double delay)
```

Constructs a new event and inserts it in the event list. If `delay >= 0.0`, the event is scheduled to happen in `delay` units of simutated time. If two or more events are scheduled to happen at the same time, they are placed in the event list (and executed) in the same order as they have been scheduled.

We recall that such constructors with parameters are not inherited automatically by the subclasses in Java, but they can be invoked using `super`. For example, one can have

```
class Bang extends Event {
   public Bang (double delay) { super (delay); }
   public void actions () {  ...  }
```

and then invoke the constructor "`new Bang(10.0)`" to get a `Bang` in 10 units of time. This is equivalent to "`new Bang().schedule(10.0)`."

Methods

`public void schedule (double delay)`

Schedules this event to happen in `delay` time units, i.e., at time `Sim.time() + delay`, by inserting it in the event list. When two or more events are scheduled to happen at the same time, they are placed in the event list (and executed) in the same order as they have been scheduled.

`public void scheduleNext()`

Schedules this event as the *first* event in the event list, to be executed at the current time (as the next event).

`public void scheduleBefore (Event other)`

Schedules this event to happen just before, and at the same time, as the event `other`. For example, if `Bing` and `Bang` are `Event` subclasses, after the statements

```
Bang bigOne = new Bang ().schedule (12.0);
new Bing ().scheduleBefore (bigOne);
```

the event list contains two new events scheduled to happen in 12 units of time: a `Bing` event, followed by a `Bang` called `bigOne`.

`public void scheduleAfter (Event other)`

Schedules this event to happen just after, and at the same time, as the event `other`.

`public void reschedule (double delay)`

Cancels this event and reschedules it to happen in `delay` time units.

`public boolean cancel()`

Cancels this event before it occurs. Returns `true` if cancellation succeeds (this event was found in the event list), `false` otherwise.

`public final boolean cancel (String type)`

Finds the first occurence of an event of class "type" in the event list, and cancels it. Returns `true` if cancellation succeeds, `false` otherwise.

`public final double time()`

Returns the (planned) time of occurence of this event.

`public abstract void actions();`

This is the method that is executed when this event occurs. Every subclass of `Event` that is to be instantiated must provide an implementation of this method.

Continuous

This abstract class provides the basic structures and tools for continuous-time simulation, where certain variables evolve continuously in time, according to differential equations. Such continuous variables can be mixed with events and processes.

Each type of continuous-time variable should be defined as a subclass of `Continuous`. The instances of these subclasses are the actual continuous-time variables. Each subclass must implement the method `deriv` which returns its derivative with respect to time. The trajectory of this variable is determined by integrating this derivative. The subclass may also reimplement the method `actions`, which is executed immediately after each integration step. By default (in the class `Continuous`), this method does nothing. This method could, for example, verify if the variable has reached a given threshold, or update a graphical illustration of the variable trajectory.

The method `selectIntegMethod` must be called before starting any integration.

```
public abstract class Continuous
```

```
public static final int EULER       = 1;
public static final int RUNGEKUTTA2 = 2;
public static final int RUNGEKUTTA4 = 4;
```

Integration methods. `EULER` means the Euler method, `RUNGEKUTTA2` and `RUNGEKUTTA4` mean the Runge-Kutta methods of orders 2 and 4, respectively.

Constructors

`public Continuous ()`

Constructs a new continuous-time variable, *without* initializing it.

`public Continuous (String name)`

Constructs a new continuous-time variable (same as `Continuous()`) with descriptor `name`. This descriptor can be used to identify the `Continuous` variable in traces and reports.

Methods

`public void init (double initval)`

Initializes or reinitializes the continuous-time variable to `initval`.

`public double value ()`

Returns the current value of this continuous-time variable.

`public static void selectIntegMethod (int method, double h)`

Selects the integration method to be used for *all* continuous-time variables (`method`), and the integration step size `h`, in time units. An integration step at time t changes the values of the variables from their old values at time $t - h$ to their new values at time t.

`public void startInteg ()`

Starts the integration process that will change the state of this variable at each integration step.

`public void startInteg (double val)`

Same as `startInteg`, after initializing the variable to `val`.

`public void stopInteg ()`

Stops the integration process for this continuous variable. The variable keeps the value it took at the last integration step before calling `stopInteg`.

`public abstract double deriv (double t);`

This method should return the derivative of this variable with respect to time, at time t. Every subclass of `Continuous` that is to be instantiated must implement it. If the derivative does not depend explicitly on time, t becomes a dummy parameter. Internally, the method is used with t not necessarily equal to the current simulation time.

`public void afterEachStep ()`

This method is executed after each integration step for this `Continuous` variable. Here, it does nothing, but every subclass of `Continuous` may reimplement it.

Process

This abstract class provides process scheduling tools. Each type of process should be defined as a subclass of the class `Process`, and must provide an implementation of the method `actions` which describes the life of a process of this class. Whenever a process instance starts, its `actions` method begins executing.

A process can be created and scheduled simultaneously by the constructor, or constructed first and scheduled later, just like an event. Scheduling a process is equivalent to placing an event in the event list that will start this process when the simulation clock reaches the specified time.

A process can be in one of the following states: `INITIAL`, `EXECUTING`, `DELAYED`, `SUSPENDED`, and `DEAD` (see the diagram). At most *one* process can be in the `EXECUTING` state at any given time, and when there is one, this executing process (called the *current process*) is said to *have control* and is executing one of the instructions of its `actions` method.

A process that has been constructed but not yet scheduled is in the `INITIAL` state. A process is in the `DELAYED` state if there is a planned event in the event list to activate it (give it control). When a process is scheduled, it is placed in the `DELAYED` state. It is in the `SUSPENDED` state if it is waiting to be reactivated (i.e., obtain control) without having an event to do so in the event list. A process can suspends itself by calling `suspend` directly or indirectly (e.g., by requesting a busy `Resource`). Usually, a `SUSPENDED` process must be reactivated by another process or event via the `resume` method. A process in the `DEAD` state no longer exists.

```
public class Process
```

```
   public static final int INITIAL   = 0;
   public static final int EXECUTING = 1;
   public static final int DELAYED   = 2;
   public static final int SUSPENDED = 3;
   public static final int DEAD      = 4;
```

The possible states of a `Process`. `INITIAL`: this process has been created but not yet scheduled; `EXECUTING`: this process is the one currently executing; `DELAYED`: this process has an event in the event list to reactivate it later on; `SUSPENDED`: this process will have to be reactivated by another process or event later on; `DEAD`: this process has terminated.

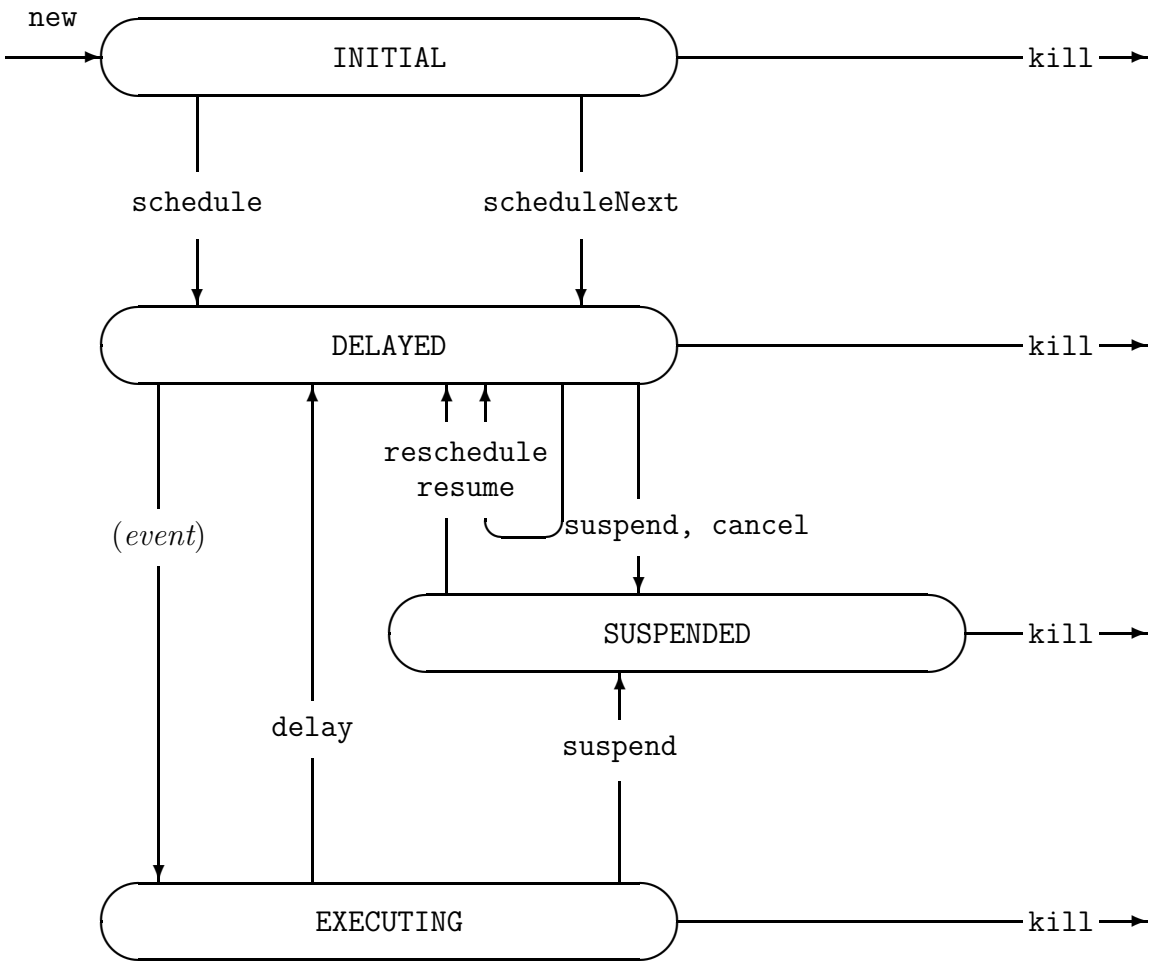
Variables

Constructors

```
public Process ()
```

Constructs a new process without scheduling it. It will have to be scheduled later on. The process is in the `INITIAL` state.

Figure 16: The possible states and state transitions of a `Process`



`public Process (double delay)`

Same as `Process().schedule(delay)`. Constructs the process as in `Process()` and also schedules it to start executing in `delay` units of time, if `delay >= 0.0`, by inserting its activating event in the event list. The process is in the `DELAYED` state.

Methods

`public void schedule (double delay)`

Schedules the process to start in `delay` time units. This puts the process in the `DELAYED` state.

`public void scheduleNext ()`

Schedules this process to start at the current time, by placing it at the beginning of the event list. This puts the process in the `DELAYED` state.

`public static Process currentProcess ()`

Returns the process that is currently executing, if any. Otherwise, returns `null`.

`public final boolean isAlive ()`

Returns true if the process is alive, otherwise false.

`public int getState ()`

Returns the state of the process.

`public double getDelay ()`

If the process is in the `DELAYED` state, returns the remaining time until the planned occurence of its activating event. Otherwise, stops the program with an error message.

`public void delay (double delay)`

Suspends the execution of the currently executing process and schedules it to resume its execution in `delay` units of simulation time. It becomes in the `DELAYED` state. Only the process in the `EXECUTING` state can call this method.

`public void reschedule (double delay)`

If the process is in the `DELAYED` state, removes it from the event list and reschedules it in `delay` units of time. Example: If the process `p` has called `delay (5.0)` at time 10.0, and another process invokes `reschedule (p, 6.2)` at time 13.5, then the process `p` will resume at time $13.5 + 6.2 = 19.7$.

`public final void suspend ()`

This method can only be invoked for the `EXECUTING` or a `DELAYED` process. If the process is `EXECUTING`, suspends execution. If the process is `DELAYED`, cancels its activating event. This places the process in the `SUSPENDED` state.

`public void resume ()`

Places this process at the beginning of the event list to resume its execution. If the process was `DELAYED`, cancels its earlier activating event.

`public boolean cancel ()`

Cancel the activating event that was supposed to resume this process, and place the process in the `SUSPENDED` state. This method can be invoked only for a process in the `DELAYED` state.

`public void kill()`

Terminates the life of this process and sets its state to `DEAD`, after cancelling its activating event if there is one.

`public static void killAll ()`

Kills all instances of the class `Process`.

Resource

Objects of this class are resources having limited capacity, and which can be requested and released by `Process` objects. These resources act indirectly as synchonization devices for processes.

A resource is created with a finite capacity, specified when invoking the `Resource` constructor, and which can be changed later on. A resource also has an infinite-capacity queue (waiting line) and a service policy that can be specified at creation (the default is FIFO).

A process must ask for a certain number of units of the resource (`request`), and obtain it, before using it. When it is done with the resource, the process must release it so that others can use it (`release`). A process does not have to request [release] all the resource units that it needs by a single call to the `request` [`release`] method. It can make several successive requests or releases, and can also hold different resources simultaneously.

Each resource maintains two lists: one contains the processes waiting for the resource (the waiting queue) and the other contains the processes currently holding units of this resource. The methods `waitList` and `servList` permit one to access these two lists. These lists actually contain objects of the class `UserRecord` instead of `Process` objects. Each `UserRecord` contains the process and some additional information.

```
public class Resource
```

```
public static final int FIFO  = 1;
public static final int LIFO  = 2;
```

Different service policies for ordering the queue (waiting list) of processes waiting for a `Resource`. `FIFO` (first in, first out): the processes are placed in the list (and served) according to their order of arrival; `LIFO` (last in, first out): the processes are placed in the list (and served) according to their inverse order of arrival, the last arrived are served first;

Constructors

```
public Resource (int capacity)
```

Constructs a new resource, with initial capacity `capacity`, service policy `FIFO`.

```
public Resource (int capacity, String name)
```

Constructs a new resource, with initial capacity `capacity`, service policy `FIFO`, and identifier `name`.

```
public Resource (int capacity, int policy, String name)
```

Constructs a new resource, with initial capacity `capacity`, service policy `policy`, and identifier `name`. The capacity can be changed later on by calling `ChangeCapacity`.

Methods

`public void collectStat (boolean flag)`

If `flag = true`, this method starts collecting statistics on the lists returned by `waitList` and `servList` for this resource. It also constructs (if not yet done) and initializes three additional statistical collectors for this resource. These collectors will be updated automatically. They can be accessed via `statOnCapacity`, `statOnUtil`, and `statOnSojourn`, respectively. The first two, of class `Accumulate`, monitor the evolution of the capacity and of the unitization (number of units busy) of the resource as a function of time. The third one, of class `Tally`, collects statistics on the processes's sojourn times (wait + service); it samples a new value each time a process releases all the units of this resource that it holds (i.e., when its `UserRecord` disappears). If `flag = false`, this method stops the collecting of statistics.

`public void initStat()`

Reinitializes all the statistical collectors for this resource. These collectors must exist, i.e., `collectStat (true)` must have been invoked earlier for this resource.

`public void init()`

Reinitializes this resource by clearing up its waiting list and service list. The processes that were in these lists (if any) remain in the same states. If statistical collection is "on", `initStat` is invoked as well.

`public int getCapacity ()`

Returns the current capacity of the resource.

`public void changeCapacity (int diff)`

Modifies by `diff` units (increases if `diff > 0`, decreases if `diff < 0`) the capacity (i.e., the number of units) of the resource. If `diff > 0` and there are processes in the waiting list whose request can now be satisfied, they obtain the resource. If `diff < 0`, there must be at least `diff` units of this resource available, otherwise the program stops with an error message (this is not a strong limitation, because one can check first and release some units, if needed, before invoking `changeCapacity`). In particular, the capacity of a resource can never become negative.

`public void setCapacity (int newcap)`

Sets the capacity to `newcap`. Equivalent to `changeCapacity (newcap - old)` if `old` is the current capacity.

`public int getAvailable()`

Returns the number of available units, i.e., the capacity minus the number of units busy.

`public void request (int n)`

The executing process invoking this method requests for `n` units of this resource. If there is enough units available to fill up the request immediately, the process obtains the desired number of units and holds them until it invokes `release` for this same resource. The process is also inserted into the `servList` list for this resource. On the other hand, if there is less than `n` units of this resource available, the executing process is placed into the `waitList` list

(the queue) for this resource and is suspended until it can obtain the requested number of units of the resource.

`public void release (int n)`

The executing process that invokes this method releases `n` units of the resource. If this process is holding exactly `n` units, it is removed from the service list of this resource and its `UserDossier` objects disappears. If this process is holding less than `n` units, the program stops with an error message. If there are other processes waiting for this resource whose requests can now be satisfied, they obtain the resource.

`public List waitList()`

Returns the list that contains the `UserRecord` objects for the processes in the waiting list for this resource.

`public List servList()`

Returns the list that contains the `UserRecord` objects for the processes in the service list for this resource.

`public Accumulate statOnCapacity()`

Returns the statistical collector that measures the evolution of the capacity of the resource as a function of time. This collector exists only if `collectStats(true)` has been invoked previously.

`public Accumulate statOnUtil()`

Returns the statistical collector for the utilization of the resource (number of units busy) as a function of time. This collector exists only if `collectStat(true)` has been invoked previously. The *utilization rate* of a resource can be obtained as the *time average* computed by this collector, divided by the capacity of the resource. The collector returned by `servList().statSize()` tracks the number of `UserRecord` objects in the service list; it differs from this collector because a process may hold more than one unit of the resource by given `UserRecord`.

`public Tally statOnSojourn()`

Returns the statistical collector for the sojourn times of the `UserRecord` objects for this resource. This collector exists only if `collectStat(true)` has been invoked previously. It gets a new observation each time a process releases all the units of this resource that it had requested by a single `request` call.

`public void report()`

Prints a complete statistical report on this resource. The method `collectStat(true)` must have been invoked previously, otherwise no statistics have been collected. The report contains statistics on the waiting times, service times, and waiting times for this resource, on the capacity, number of units busy, and size of the queue as a function of time, and on the utilization rate.

Class

`class UserRecord`

Each time a process requests for a `Resource`, a record (an instance of `UserRecord`) is created to store the information relative to that request. The lists `waitList` and `servList` contains objects of the class `UserRecord`.

Bin

A `Bin` corresponds to a pile of identical tokens, and a list of processes waiting for the tokens when the list is empty. It is a producer/consumer process synchronization device. Tokens can be added to the pile (i.e., *produced*) by the method `put`. A process can request tokens from the pile (i.e., *consume*) by calling `take`.

The behavior of a `Bin` is somewhat similar to that of a `Resource`. Each `Bin` has a single queue of waiting processes, whose service policy is specified at creation, and which can be accessed via the method `waitList`. This list actually contains objects of the class `UserRecord`. Each `UserRecord` points to a process and contains some additional information.

```
public class Bin
```

```
public static final int FIFO  = 1;
public static final int LIFO  = 2;
```

Service policies available for ordering the queue (waiting list) of processes waiting for tokens from a `Bin`. `FIFO` (first in, first out): the processes are placed in the list (and served) according to their order of arrival; `LIFO` (last in, first out): the processes are placed in the list (and served) according to their inverse order of arrival, the last arrived are served first;

Constructors

```
public Bin (String name)
```

Constructs a new bin, initially empty, with service policy `FIFO` and identifier `name`.

```
public Bin (int policy, String name)
```

Constructs a new bin, initially empty, with service policy `policy` and identifier `name`.

Methods

```
public void init()
```

Reinitializes this bin by clearing up its pile of token and its waiting list. The processes in this list (if any) remain in the same states.

```
public int getAvailable()
```

Returns the number of available tokens for this bin.

```
public void take (int n)
```

The executing process invoking this method requests `n` tokens from this bin. If enough tokens are available, the number of tokens in the bin is reduced by `n` and the process continues its execution. Otherwise, the executing process is placed into the `waitList` list (the queue) for this bin and is suspended until it can obtain the requested number of tokens.

```
public void put (int n)
```

Adds `n` tokens to this bin. If there are processes waiting for tokens from this bin and whose requests can now be satisfied, they obtain the tokens and resume their execution.

```
public List waitList()
```

Returns the list of `UserRecord` objects for the processes waiting for tokens from this bin.

Class

```
public class UserRecord {
   public Process user ()
   public int nbTokens ()
}
```

Each time a process requests tokens from a `Bin` and must be put into the queue because not enough tokens are available, a `UserRecord` is created to store the information on that request, and is inserted into the list returned by `waitList`. The methods `user` and `nbTokens` return the waiting process and the number of requested tokens for this record.

Condition

A `Condition` is a boolean indicator, with a list of processes waiting for the indicator to be `true` (when it is `false`). A process calling `wait` on a condition that is currently `false` is suspended until the condition becomes `true`. The list of waiting processes can be accessed via `waitList`.

```
public class Condition
```

Constructor

`public Condition (boolean val, String name)`

Constructs a new `Condition` with initial value `val` and identifier `name`.

Methods

`public void init (boolean val)`

Reinitializes this `Condition` by clearing up its waiting list and resetting its state to `val`. The processes in this list (if any) remain in the same states.

`public void set (boolean val)`

Sets the condition to `val`. If `val` is `true`, all the processes waiting for this condition now resume their execution, in the same order as they have called `wait` for this condition. (Note that a process can never wait for more than one condition at a time, because it cannot call `wait` while it is suspended.

`public boolean state ()`

Returns the state (`true` or `false`) of the condition.

`public void waitFor ()`

The executing process invoking this method must wait for this condition to be `true`. If it is already `true`, the process continues its execution immediately Otherwise, the executing process is placed into the `waitList` list for this condition and is suspended until the condition becomes `true`.

`public List waitList()`

Returns the list of `UserRecord` objects for the processes waiting for this condition.

Class

```
public class UserRecord {

   public Process user ()
}
```

Each time a process must wait for a `Condition`, a `UserRecord` is created to store the information on that request, and is inserted into the list returned by `waitList`. The method `user` returns the waiting process for this record.

EventList

An interface for implementations of future event lists. Different implementations are provided in SSJ: doubly linked list, splay tree, Henrickson's method, etc. The *events* in the event list are objects of the class `Event`. The method `Sim.init` permits one to select the actual implementation used in a simulation.

```
public interface EventList
```

`public boolean isEmpty ();`

Returns `true` if and only if the event list is empty (no event is scheduled).

`public void cleanup ();`

Empties the event list, i.e., cancels all events.

`public void print ();`

Prints the contents of the event list.

`public void insert (Event ev);`

Inserts a new event in the event list, according to the field `eventTime` of `ev`.

`public void insertFirst (Event ev);`

Inserts a new event notice at the beging of the event list.

`public void insertBefore (Event ev, Event other);`

Same as `insert`, but inserts the new event `ev` immediately before the event `other` in the list.

`public void insertAfter (Event ev, Event other);`

Same as `insert`, but inserts the new event `ev` immediately after the event `other` in the list.

`public Event viewFirst ();`

Returns the first event in the event list.

`public Event viewFirstOfClass (String cl);`

Returns the first event of the class `cl` (a subclass of `Event`) in the event list.

`public boolean remove (Event ev);`

Removes the event `ev` from the event list (cancels this event).

`public Event removeFirst ();`

Removes the first event from the event list (to cancel or execute this event).

DoublyLinked

An implementation of `EventList` using a doubly linked linear list.

```
public class DoublyLinked implements EventList
```

Constructor

`public DoublyLinked ()`

Constructs a new doubly linked linear list.

Methods

`public boolean isEmpty ()`

`public void print ()`

`public Node findPosition (Event ev)`

Finds the position where `ev` should be inserted in the event list, according to its field `eventTime`, starting from the end of the list. Returns the node *after which* this event should be inserted.

`public void insert (Event ev)`

`public void insertFirst (Event ev)`

`public void insertBefore (Event ev, Event other)`

`public void insertAfter (Event ev, Event other)`

`public Event viewFirst ()`

`public Event viewFirstOfClass (String cl)`

`public boolean remove (Event ev)`

`public Event removeFirst ()`

References

[1] J. Banks, J. S. Carson, B. L. Nelson, and D. M. Nicol. *Discrete-Event System Simulation*. Prentice-Hall, Upper Saddle River, N.J., third edition, 2000.

[2] G. M. Birtwistle. *Demos Reference Manual*. Dep. of Computer Science, University of Bradford, Bradford, 1979.

[3] G. M. Birtwistle, G. Lomow, B. Unger, and P. Luker. Process style packages for discrete event modelling. *Transactions of the Society for Computer Simulation*, 3-4:279–318, 1986.

[4] P. Bratley, B. L. Fox, and L. E. Schrage. *A Guide to Simulation*. Springer-Verlag, New York, second edition, 1987.

[5] G. S. Fishman. *Monte Carlo: Concepts, Algorithms, and Applications*. Springer Series in Operations Research. Springer-Verlag, New York, 1996.

[6] W. J. Kennedy Jr. and J. E. Gentle. *Statistical Computing*. Dekker, New York, 1980.

[7] L. Kleinrock. *Queueing Systems, Vol. 1*. Wiley, New York, 1975.

[8] W. Kreutzer. *System Simulation - Programming Styles and Languages*. Addison Wesley, New York, 1986.

[9] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, New York, third edition, 2000.

[10] P. L'Ecuyer. SIMPascal 2.0: Guide de l'usager pour VAX/VMS. Technical Report DIUL-RT-8705, Département d'informatique, Université Laval, 1987.

[11] P. L'Ecuyer. SIMOD: Définition fonctionnelle et guide d'utilisation (version 2.0). Technical Report DIUL-RT-8804, Département d'informatique, Université Laval, Sept 1988.

[12] P. L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1):159–164, 1999.

[13] P. L'Ecuyer and S. Côté. Implementing a random number package with splitting facilities. *ACM Transactions on Mathematical Software*, 17(1):98–111, 1991.

[14] P. L'Ecuyer and N. Giroux. A process-oriented simulation package based on Modula-2. In *1987 Winter Simulation Proceedings*, pages 165–174, 1987.

[15] P. L'Ecuyer, R. Simard, E. J. Chen, and W. D. Kelton. An object-oriented random-number package with many long streams and substreams. *Operations Research*, 2002. To appear.

[16] R. W. Wolff. *Stochastic Modeling and the Theory of Queues*. Prentice-Hall, New York, 1989.