Proceedings of the 2002 Winter Simulation Conference
E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, eds.

SSJ: A FRAMEWORK FOR STOCHASTIC SIMULATION IN JAVA

Pierre L'Ecuyer
Lakhdar Meliani
Jean Vaucher

Département d'Informatique et de Recherche Opérationnelle
Université de Montréal, C.P. 6128, Succ. Centre-Ville
Montréal, H3C 3J7, CANADA

ABSTRACT

We introduce SSJ, an organized set of software tools implemented in the Java programming language and offering general-purpose facilities for stochastic simulation programming. It supports the event view, process view, continuous simulation, and arbitrary mixtures of these. Performance, flexibility, and extensibility were key criteria in its design and implementation. We illustrate its use by simple examples and discuss how we dealt with some performance issues in the implementation.

1 INTRODUCTION

SSJ is a Java-based framework for simulation programming. It has been designed primarily for discrete-event stochastic simulations (Law and Kelton 2000), but it also supports continuous and mixed simulation. Some ideas in its design are inherited from the packages DEMOS (Birtwistle 1979) (based on the *Simula* language) and SIMOD (L'Ecuyer and Giroux 1987) (based on the *Modula-2* language), among others.

Simulation models can be implemented in a variety of languages, including general-purpose programming languages such as FORTRAN, C, C++, Java, etc., specialized simulation languages such as GPSS, SIMAN, SLAM, SIMSCRIPT, etc., and point-click-drag-drop graphical simulation environments such as Arena, Automod, etc.

Specialized languages and environments provide higher-level tools but usually at the expense of being less flexible than general purpose programming languages. In commercial simulation languages and environments, one must frequently revert to general-purpose languages (such as C or VisualBasic) to program the more complex aspects of a model or unsupported operations. Compilers and supporting tools 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-drop operations, as in computer games. This approach is very handy for building models that happen to fit the frameworks pre-programmed in the software. But for large and complex 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 reputed to be highly portable. 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 certain 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 "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 complex systems (Franta 1977, Birtwistle et al. 1986, Kreutzer 1986, Law and Kelton 2000). On the other hand, certain systems are more conveniently modeled simply with *events*, which are instantaneous in the simulation time frame. Sometimes, the use of events is preferred because it gives a faster sim-

ulation program, by avoiding the process-synchronization overhead. Events and processes can be mixed freely in SSJ.

Other Java-based simulation frameworks and libraries proposed in the last few years include *Silk* (Healy and Kilgore 1997, Kilgore 2000), a *SimJava* from New Zealand (Kreutzer, Hopkins, and van Mierlo 1997), a *simjava* from Scotland (Howell and McNab 1998), a *JavaSim* from the U.K. (Little 1999), a *JavaSim* from Ohio (Tyan and Hou 2002), *JSIM* (Miller, Ge, and Tao 1998), and *Simkit* (Buss 2000, 2001). Some of them are specialized (e.g., for queueing systems, computer networks and protocols, etc.). Our framework has a different design, in several aspects, than each of these. We won't discuss them all, but for illustration we will make some timing comparisons with *Silk*, which is perhaps the best known of these products in the WSC community.

SSJ was designed to be open and very flexible from bottom to top. As an example, at the lower level, the event list has a default implementation and the user need not worry about it, but it is also easy to change the type of data structure used for its implementation (doubly linked list, binary tree, heap, splay tree, etc.), by adding one extra parameter to the "clock and event list initialization" method. As another example, different types of uniform random number generation algorithms are available in SSJ, and random *streams* (which are objects acting roughly like independent random number generators) can be created at will using any of these types. Some of these algorithms may correspond to quasi-Monte Carlo methods, i.e., they produce (deterministic or randomized) low-discrepancy sequences (see, e.g., L'Ecuyer and Lemieux 2002) instead of pseudo-random numbers. This provides a nice way of mixing Monte Carlo and quasi-Monte Carlo methods in a given simulation program, and switching between them, without changing the simulation code: It suffices to change the constructor at stream creation to change the type of stream. Convenient tools are also available to manipulate and control the streams in order to facilitate the implementation of variance reduction methods that require some form of synchronization (L'Ecuyer et al. 2002, Law and Kelton 2000).

We have been attentive to performance issues. Execution speed remains important for many (if not most) serious simulation applications. In some cases (e.g., when pricing financial derivatives by simulation) it is because precise estimates are required within a few seconds or minutes; in other cases (e.g., when optimizing complex stochastic systems by simulation) a well-tuned efficient program may already run for several hours or even a few days, so slowing it down by a factor of 10 (say) makes a significant difference. The constant increase of cheap computing power will not change this state of affairs in the foreseeable future: People just adapt by considering larger and more detailed models, and by attacking more difficult optimization problems.

Java has a bad reputation for execution speed, so if performance is important, why choose it? It is true that the early versions of the Java Virtual Machine (JVM), which interprets the portable byte code, were slow. But things have changed dramatically with the most recent versions. The JVMs are now optimized and complemented with just-in-time compilers. In these environments, Java programs can run almost as fast as C or C++ programs compiled in native code. Given the fact that simulation programs in Java are much more elegant, clean, and portable than their counterparts in C, the small "speed tax" is worth the price.

The remainder of this paper is organized as follows. The next section gives a brief overview of SSJ. To give a taste of how simulation programs look like with SSJ, we develop and discuss small examples in Section 3. Section 4 explains how we have implemented the processes by harnessing Java threads. Section 5 outlines intended future developments. A complete documentation of all the classes provided by SSJ so far, and several additional examples, are given by L'Ecuyer (2001b) and Meliani (2002).

2 OVERVIEW OF SSJ

Low-level classes in SSJ implement basic tools such as random number generators, statistical probes, and general-purpose lists. Each class providing a uniform random number generator must implement an interface named `RandomStream`, which looks pretty much like the interface of the class `RngStream` described by L'Ecuyer (2001a) and L'Ecuyer et al. (2002), with multiple streams and substreams. Other classes provide methods for generating non-uniform random variates from several kinds of distributions.

The class `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`.

Event scheduling is managed by the class `Sim`, which contains the simulation clock and a central monitor. 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 EXAMPLES

3.1 An $M/M/1$ Queue

Our first example is a traditional $M/M/1$ queue, with arrival rate of 1 and mean service time of 0.8. The system initially starts empty. We want to simulate its operation and compute statistics such as the mean waiting time per customer, the mean queue length, etc. Simulation is not necessarily the best tool for this very simple model (queueing formulas are available for infinite-horizon averages; see, e.g., Kleinrock 1975), but we find it convenient for illustrating SSJ.

Figure 1 shows a SSJ-based process-oriented simulation program that simulates the $M/M/1$ queue for one million time units (i.e., approximately one million customers). The constants `meanArr`, `meanServ`, and `timeHorizon` represent the mean time between arrivals, the mean service time, and the time horizon, respectively. The server is an object of class `Resource`, with capacity 1. The two random number streams `genArr` and `genServ` (instances of the class `RandMrg`) are used to generate the interarrival times and service times, respectively. These objects are created when `QueueProc` is instantiated by the `main` program.

The program defines one type of process (`Customer`) by extending the pre-defined class `Process`. The method `actions` (which must be implement by any extension of `Process`) describes the life of a customer. Upon arrival, the customer first schedules the arrival of the next customer in an exponential number of time units. Behind the scenes, this effectively schedules an event, in the event list, that will start a new customer instance. 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, it 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 starts, the customer invokes `delay` to freeze itself for a duration equal to its exponential service time. After this delay has elapsed, the customer releases the server and disappears. Several distinct customer instances can co-exist in the simulation at any given point in time, and be at different phases of their `actions` method.

The program also defines one type of event (`EndOfSim`) by extending the class `Event` and defining its method `actions`. This method describes what to do when this event occurs (at the end of the simulation).

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 10^6, schedules the first customer's arrival, and starts the simulation. The `EndOfSim` event prints a detailed statistical report on the resource `server` (average utilization, average waiting time, average queue length, number of customers served, etc.).

One can also write an event-oriented version of this $M/M/1$ queue simulation program, where the event classes are `Arrival`, `Departure`, and `EndOfSim`, as shown in Figure 2. This program is written at a lower level and is less compact than its process-oriented counterpart. On the other hand, it runs faster (see below). This is often true, due to the fact that processes involve more overhead. Working at a lower level is also convenient in situations where the logic implemented in the available higher-level constructs is not exactly what we want.

Here, the customers waiting and in service are maintained in lists `waitList` and `servList`. The statistical probe `custWaits` collects statistics on the customer's waiting times. It is of class `Tally`, which is appropriate when the statistical data of interest is a sequence of observations $X_1, X_2, \ldots$. Every call to `waitList.update` brings a new observation X_i (a new customer's waiting time). The statistical probe `totWait`, of class `Accumulate`, computes the integral (and eventually the time-average) of the queue length as a function of time. It is updated each time the queue size changes. Interestingly, in the program of Figure 1, the `Resource` and `Process` objects use these same lower-level constructs (lists, statistical probes, etc.), but this is hidden in SSJ.

Note that in this program, only one `Arrival` event and one `Departure` event are instantiated. These events are recycled. This contributes to improving the speed by reducing the number of objects that must be created. This is allowed because there is never more than one instance of these events planned at the same time. Of course, this program could be further simplified in several ways. In fact, the successive customer's waiting times can be simulated without an event list by using Lindley's recurrence $W_{i+1} = \max(0,\ W_i + S_i - A_i)$ where A_i, S_i, and W_i are the ith interarrival time, service time, and waiting time, respectively. Our goal here is not to make the simplest program for the $M/M/1$ queue, but to illustrate SSJ.

We made some timing experiments with these two programs on a 750 MHz AMD-Athlon computer running Redhat Linux 7.1. With the JDK-1.2.2 virtual machine, the `javacomp` just-in-time compiler, and Java green threads,

```
public class QueueProc {

   static final double meanArr   = 1.0;
   static final double meanServ   = 0.8;
   static final double timeHorizon = 1000000.0;

   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 () {
         System.out.println(server.report());
         Sim.stop();
         }
      }
   }
```

Figure 1: Process-oriented Simulation of an $M/M/1$ Queue.

`QueueProc` took 21.9 seconds to run and `QueueEv2` took 7.1 seconds. These numbers are the "user time" returned by the Linux command "`time`". With JDK-1.3.1 and the "hotspot" optimizer from SUN, `QueueEv2` took 3.6 seconds. However, the hotspot optimizer allows only *native threads* (which are real threads managed at the operating system level, as opposed to *green* threads which are simulated threads in the Java environments). `QueueProc` runs much slower under this setup (over a minute), because native threads involve a large amount of overhead. A program that simply implements Lindley's recurrence to compute the average waiting time of 10^6 customers takes approximately 2.1 seconds under JDK-1.3.1 + hotspot, and more than 3/4 of this time is used to generate the exponential random variables.

A *Silk* version of `QueueProc`, taken from Healy and Kilgore (1997) with straightforward adaptation, took 190 seconds under JDK-1.2.2 with `javacomp` using Silk's academic version 1.2.

SSJ actually has a brother named SSC (L'Ecuyer 2002). It is a library implemented in ANSI-C, offering tools similar to those of SSJ, but without the process-view facilities. We tried a version of `QueueEv2` in SSC and it ran in 2.8 seconds on the same machine (with `gcc` and optimization level `-O3`). This gives a Java/C speed ratio of $3.6/2.8 \approx 1.3$, i.e., a time penalty of approximately 30%.

3.2 Two Queues in Series with a Batch Server

This example is adapted from Healy and Kilgore (1998). Customers must pass through two queues in series: After being served at the first queue, they join the second queue, and when their service is over at the second queue, they disappear. The first queue is $M/M/1$ as in the previous example while the second one serves customers in batches of sizes 10 to 20. The server at the second queue waits until there are at least 10 customers ready to be served, then serves all these customers simultaneously. After completing its service time, if $C \geq 10$ customers are waiting in queue 2, the server starts another batch immediately with $\min(C, 20)$ customers. Otherwise it waits until there are 10.

```
public class QueueEv2 {

  static final double meanArr    = 1.0;
  static final double meanServ   = 0.8;
  static final double timeHorizon = 1000000.0;

  Arrival  arrival  = new Arrival();
  Departure departure = new Departure();

  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 QueueEv2(); }

  public QueueEv2() {
     Sim.init();
     new EndOfSim().schedule (timeHorizon);
     arrival.schedule (Rand1.expon (genArr, meanArr));
     Sim.start();
     }
  class Arrival extends Event {
     public void actions() {
        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.addLast (cust);
           // totWait.update (waitList.size());
        } else {                  // Starts service.
           servList.addLast (cust);
           departure.schedule (cust.servTime);
           custWaits.update (0.0);
           }
        }
     }
  class Departure extends Event {
     public void actions () {
        servList.removeFirst ();
        if (waitList.size () > 0) {
           // Starts service for next one in queue.
           Customer cust = (Customer) waitList.removeFirst ();
           servList.addLast (cust);
           departure.schedule (cust.servTime);
           custWaits.update (Sim.time () - cust.arrivTime);
           // totWait.update (waitList.size ());
           }
        }
     }
  class EndOfSim extends Event {
     public void actions () {
        System.out.println (custWaits.report());
        Sim.stop();
        }
     }
  }
```

Figure 2: Event-oriented Simulation of an $M/M/1$ Queue.

Suppose the arrival rate is 1, the service time at the first queue is exponential with mean 0.8, and the service time at the second queue is exponential with mean θ. We want to compare the mean sojourn time in the system for two different values of θ. To be specific, let X_1 (resp., X_2) be the average sojourn time of the first 10000 customers for $\theta = \theta_1 = 12$ (resp., $\theta = \theta_2 = 13$). We want to estimate $\mu_2 - \mu_1$ where $\mu_j = E[X_j]$. To do this, we will perform 10 pairs of simulation runs, where each pair produces a replicate of the vector (X_1, X_2) using common random numbers across the two values of θ (see, e.g., Law and Kelton 2000 for an introduction to this variance reduction technique). The 10 replicates of $D = X_2 - X_1$ are independent random variables that are approximately normally distributed, so we can compute a confidence interval on $\mu_2 - \mu_1$ by assuming that $\sqrt{10}(\bar{D} - (\mu_2 - \mu_1))/S_D$ has the Student distribution with 9 degrees of freedom, where $\bar{D}$ and S_D^2 are the empirical mean and variance of the 10 values of D. The program of Figure 3 computes such a confidence interval and produces the printout shown in Figure 4.

The constructor of `BatchServer` repeats the following 10 times: It performs one simulation run at $\theta = 12.0$, memorizes the average sojourn time in variable `mean1`, resets the three random number streams to the beginning of their current substreams so that the next simulation run at $\theta = 13.0$ will use exactly the same sequences of random numbers as that at $\theta = 12.0$, performs the second simulation, gives the value of $D = X_2 - X_1$ to the collector `statDiff`, and then resets the random number streams to new substreams in order to get independent random numbers for the next pair of runs. When the 10 pairs of runs are completed, the 90% confidence interval on $\mu_2 - \mu_1$ is printed.

Here, the server at the first queue is implemented as a `Resource` as in the $M/M/1$ example, but the second server is implemented as a `Process`, using a `Bin` as a synchronization mechanism. When a customer arrives at queue 2, if he is the tenth customer in the queue and the server is free, he wakes up the server. The customer then requests a token from that bin and is automatically suspended until he receives the token (when its service ends). The method `binServ2.waitList().size()` returns the number of processes currently waiting for tokens at the bin; this is the size of the queue at the second server.

The behavior of the second server is described in the `actions` method of `Server2`. When this server becomes free and the queue size is less than 10, it suspends itself, waiting to be waken up by a customer when the queue size reaches 10. The server then becomes busy and serves a new batch of customers. The size of this batch may exceed 10 if the server just completed the previous batch (and was not suspended). After service completion, a number of tokens equal to the batch size is put on the bin, so that the appropriate number of customers can resume their execution. Note that the method `Sim.init()` automatically cleans up all `Process` objects; for this reason, `server2` must be created at the beginning of each simulation run.

In contrast to the previous examples, here there is no `EndOfSim` event; customer number 10000 takes care of stopping the simulation when he leaves the system.

Under JDK-1.2.2 with `javacomp`, this program takes about 7.8 seconds to execute. We tried the *Silk* implementation of the same model, given in Healy and Kilgore (1998), and it took approximately 100 seconds for a *single* run (compared with roughly 0.4 seconds per run with SSJ) on the same platform.

4 IMPLEMENTING PROCESSES

In discrete event simulation, sequencing is based on an ordered list of event notices. In SSJ, these notices are represented by user defined sub-classes of the `Event` class, each of which must contain an `actions` method defining the effect of the event. A simulation executive repeatedly takes the next event off the list, updates the simulation clock and executes the `actions` method until the end.

SSJ also handles Processes. These act like threads: they execute a sequence of events spread over time so that their actions are intermingled with those of other processes. In Java, it seems natural to use the `Thread` class to implement Processes. However, Java `Thread`s are designed to support *real* parallelism (exploiting multi-processor architectures) or pseudo parallelism via time-slicing. For simulation, one must find an efficient way to curb this parallelism so that execution passes between the executive and the processes in a strictly sequential way.

The required control operation is the `resume(X)` operation of *coroutines* where the active thread suspends itself and passes control to another thread so that there is always only one thread active. Presently, Java provides a version of `resume(X)` which does not automatically suspend the calling thread so that the `resume(X);` must be followed by `suspend;`. To allow for combined *event* and *process* orientations, the `actions` method of process objects just *resumes* the process and the processes *resume* the executive after every active phase.

Unfortunately, as Java implementations have evolved, the use of `resume(X)` and `suspend` has been deprecated: these operations may not be available in future versions. We have also found that correct passage of control cannot be garanteed. In SSJ, process synchronization does not use Java's `resume(X)`; rather, we use the `wait` and `notify` methods to implement semaphores (Holub 2000) and use those to create our own safe and correct versions of `resume(X)` and `suspend`. These were the ones used in the `BatchServer` example.

To deal with real—not simulated—parallelism, newer implementations of Java use operating systems *native*

```
class BatchServer {

  static final double meanArr  = 1.0;
  static final double meanServ1 = 0.8;
  static double meanServ2;

  RandMrg  genArr   = new RandMrg();  // For times between arrivals.
  RandMrg  genServ1  = new RandMrg();  // For service times at server 1.
  RandMrg  genServ2  = new RandMrg();  // For service times at server 2.

  Resource server1  = new Resource (1, "Server 1");
  Server2  server2;
  Bin    binServ2  = new Bin ("Server 2 Bin ");
  Tally   statSojourn = new Tally ("Sojourn times in one run");
  Tally   statDiff   = new Tally ("Differences on averages");

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

  public BatchServer () {
     for (int rep = 0; rep < 10; rep++) {
        meanServ2 = 12.0;  simulOneRun();
        double mean1 = statSojourn.average();
        genArr.resetStartSubstream ();
        genServ1.resetStartSubstream ();
        genServ2.resetStartSubstream ();
        meanServ2 = 13.0;  simulOneRun();
        statDiff.update (statSojourn.average() - mean1);
        genArr.resetNextSubstream ();
        genServ1.resetNextSubstream ();
        genServ2.resetNextSubstream ();
     }
     System.out.println (statDiff.printConfIntStudent (0.90));
  }
  private void simulOneRun () {
     statSojourn.init();  server1.init();  binServ2.init();
     Sim.init();    // Note: this method kills all processes.
     server2 = new Server2();  server2.schedule(0.0);
     new Customer().schedule(0.0);
     Sim.start();
  }
  class Customer extends Process {
     public void actions() {
        new Customer().schedule (Rand1.expon (genArr, meanArr));
        double arrivalTime = Sim.time();
        server1.request(1);
        delay (Rand1.expon (genServ1, meanServ1));
        server1.release(1);
        if (binServ2.waitList().size() >= 9 && !server2.busy)
           server2.resume();
        binServ2.take(1);           // Blocked until end of service.
        statSojourn.update (Sim.time() - arrivalTime);
        if (statSojourn.numberObs() >= 10000) Sim.stop();
     }
  }
  class Server2 extends Process {
     boolean  busy = false;
     int    batchSize;             // Current batch size.
     public void actions() {
        while (true) {
           if (binServ2.waitList().size() < 10) {
              busy = false;  suspend(); // Wait for enough customers.
           };
           busy = true;             // Starts serving new batch.
           batchSize = Math.min (20, binServ2.waitList().size());
           delay (Rand1.expon (genServ2, meanServ2));
           binServ2.put (batchSize);    // Unblocks customers in batch.
        }
     }
  }
}
```

Figure 3: Simulation Program for Two Queues in Series with a Batch Server.

```
REPORT on Tally stat. collector ==> Differences on averages
      min      max    average   standard dev.  nb. obs
     3.373     8.050    4.958      1.644      10

 90.0% confidence interval for mean ( 4.005, 5.911 )
```

Figure 4: Output of the Program of Figure 3.

threads rather than manage their own locally. This means that scheduling overhead has increased to the point where the time required to create and start a new thread is easily the equivalent of creating 100 ordinary objects.

To improve efficiency, we do not implement processes directly as `Thread`s; rather we have an intermediate (private) class `Thread2` which execute the `actions` method of the Processes. The advantage of this organization is that we do not need to create a new `Thread` for every new `Process`. The `Thread2` objects are organized in a *thread pool* (Holub 2000). When a process ends its life, its associated `Thread2` object is put on a stack of free threads. When a new process is instantiated, an old `Thread2` object is taken from the stack, or created if the stack is empty. Thus, the total number of threads created does not exceed the maximum number of threads that are simultaneously active during the simulation. However, with native threads, the number is limited and some complex simulations can run out of threads.

Efficiency is also improved by sometimes by-passing the executive. Initially, after every event, the executive takes control and calls the `actions` method of the next `Event` or `Process`; but, when it is time for a process to relinquish control, its thread takes over from the executive, executing the upcoming events (if any) from the event list, until it comes upon another process. Then, it *resumes* directly the next process. This mechanism reduces by half the number of transfers between threads.

Nevertheless, as the `QueueProc` and `QueueEv2` timings indicate, event models which only use method calls still run roughly 3 times faster than process models which require `Thread` operations.

5 FUTURE DEVELOPMENTS AND CONCLUSION

We presented SSJ, a framework written in Java which allows both discrete and continuous simulation and supports both events and processes. Its strength lies partly in the state-of-the-art support for random number generation, efficient implementation, and use of novel scheduling techniques adapted to Java's strengths and weaknesses.

Our work allows us to draw some conclusions about the suitability of Java for simulation. For sequential programming or event-oriented simulation, the code runs almost as fast as C (30% penalty in our example). Surprisingly, Java's support for real parallel activity via the `Thread` class is ill-adapted to the pseudo-parallelism of simulation processes.

During the year 2002, we plan to beef up SSJ's library of alternate event list implementations, random number generators, support for quasi-Monte Carlo methods, statistical analysis tools, and to add classes adapted to specific areas of applications such as finance.

ACKNOWLEDGMENTS

This work has been supported by NSERC-Canada Grant No. ODGP0110050 and FCAR-Québec Grant No. 00ER3218 to the first author. We thank Guy Lapalme for his helpful comments on the design of SSJ classes.

REFERENCES

Birtwistle, G. M. 1979. *Demos—a system for discrete event modelling on Simula*. London: MacMillan.

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

Buss, A. H. 2000. Component-based simulation modeling. In *Proceedings of the 2000 Winter Simulation Conference*, 964–971. Pistacaway, New Jersey: IEEE Press.

Buss, A. H. 2001. Discrete event programming with Simkit. *Simulation News Europe* 32/33: 15–26.

Franta, W. R. 1977. *The process view of simulation*. New York: North Holland.

Healy, K. J., and R. A. Kilgore. 1997. Silk: A Java-based process simulation language. In *Proceedings of the 1997 Winter Simulation Conference*, 475–482. Piscataway, NJ: IEEE Press.

Healy, K. J., and R. A. Kilgore. 1998. Introduction to silk and Java-based simulation. In *Proceedings of the 1998 Winter Simulation Conference*, 327–334. Piscataway, NJ: IEEE Press.

Holub, A. 2000. *Taming Java threads*. APress (distributed by Springer-Verlag, NY).

Howell, F. W., and R. McNab. 1998. Simjava: a discrete event simulation package for Java with applications in

computer systems modelling. In *Proceedings of the First International Conference on Web-based Modelling and Simulation*. San Diego, CA: The Society for Computer Simulation.

Kilgore, R. A. 2000. Silk, Java, and object-oriented simulation. In *Proceedings of the 2000 Winter Simulation Conference*, 246–252. Piscataway, NJ: IEEE Press.

Kleinrock, L. 1975. *Queueing systems, vol. 1*. New York: Wiley.

Kreutzer, W. 1986. *System simulation - programming styles and languages*. New York: Addison Wesley.

Kreutzer, W., J. Hopkins, and M. van Mierlo. 1997. SimJAVA—a framework for modeling queueing networks in Java. In *Proceedings of the 1997 Winter Simulation Conference*, 483–488. Pistacaway, New Jersey: IEEE Press.

Law, A. M., and W. D. Kelton. 2000. *Simulation modeling and analysis*. Third ed. New York: McGraw-Hill.

L'Ecuyer, P. 2001a. Software for uniform random number generation: Distinguishing the good and the bad. In *Proceedings of the 2001 Winter Simulation Conference*, 95–105. Pistacaway, NJ: IEEE Press.

L'Ecuyer, P. 2001b. *SSJ: A Java library for stochastic simulation*. Software user's guide.

L'Ecuyer, P. 2002. *SSC: A library for stochastic simulation in C*. Software user's guide.

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

L'Ecuyer, P., and C. Lemieux. 2002. Recent advances in randomized quasi-monte carlo methods. In *Modeling Uncertainty: An Examination of Stochastic Theory, Methods, and Applications*, ed. M. Dror, P. L'Ecuyer, and F. Szidarovszki, 419–474. Boston: Kluwer Academic Publishers.

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

Little, M. C. 1999. JavaSim user's guide. Available on-line at <http://javasim.ncl.ac.uk/>.

Meliani, L. 2002. Un cadre d'application pour la simulation stochastique en Java. Master's thesis, DIRO, Université de Montréal. Forthcoming.

Miller, J. A., Y. Ge, and J. Tao. 1998. Component-based simulation environments: JSIM as a case study using Java beans. In *Proceedings of the 1998 Winter Simulation Conference*, 373–381. Piscataway, NJ: IEEE Press.

Tyan, H.-Y., and C.-J. Hou. 2002. JavaSim on-line manuals and tutorials. Available on-line at <http://javasim.cs.uiuc.edu>.

AUTHOR BIOGRAPHIES

PIERRE L'ECUYER is professor in the "Département d'Informatique et de Recherche Opérationnelle", at the University of Montreal. His main research interests are random number generation, quasi-Monte Carlo methods, efficiency improvement via variance reduction, sensitivity analysis and optimization of discrete-event stochastic systems, and discrete-event simulation in general. He obtained the prestigious *E. W. R. Steacie* Grant in 1995-97 and the *Killam* Grant in 2001-03. His recent research articles are available on-line at <http://www.iro.umontreal.ca/~lecuyer>.

LAKHDAR MELIANI is a M.Sc. Student at the University of Montreal. His main interests are software engineering and objected-oriented programming.

JEAN VAUCHER is professor in the "Département d'Informatique et de Recherche Opérationnelle", at the University of Montreal. In the early seventies, he designed GPSSS, a simulation package based on Simula, and made several contributions related to efficient event list implementations. Presently, his main research interest are in object-oriented programming, distributed systems and intelligent agents.