Discrete-Time Process-Oriented Simulation with J-Sim

Jaroslav Kačer[1]

University of West Bohemia, Faculty of Applied Sciences,
Department of Computer Science and Engineering,
Univerzitní 8, 30614 Plzeň, Czech Republic
jkacer@kiv.zcu.cz

Abstract. This paper describes J-Sim, a Java library for discrete-time process-oriented simulation. J-Sim is a fully portable successor to C-Sim, an already existing library written in C. The concepts used in both libraries are inherited from the Simula language. The theoretical background, basic principles of implementation, and two simple examples of use are presented in this paper.

1 Introduction

J-Sim is a software package whose primary goal is to facilitate simulation of discrete-time systems in Java. It is strongly inspired by Simula, the first widely used simulation language, developed in the 1960s by Ole-Johan Dahl and Kristen Nygaard. (More information about the history and principles of Simula can be found in [4].) J-Sim offers all concepts known from Simula, including the possibility of modelling *networks* which consist of *active stations* serving a *passive flow of customers*. Stations are usually composed of two parts: a *queue* and a *server*, managing the queue. However, a different approach can be taken: the customers can be represented by active objects, interacting with passive stations. J-Sim is completely independent of any of the two models, it is only a tool allowing a specific model to be described and simulated using the Java language. Moreover, J-Sim is not limited to queueing networks modelling, it can be used for any kind of simulation having discrete-time character.

2 The Simulation World Description

A simulation model can contain a various number of independent *active processes*. Every process has its own pre-programmed *life* which can be divided into parts. All processes within the same simulation model share the same time, called the *simulation time*. Its value is equal to zero before the simulation starts and can only be increased during its progress. One part of a process's life is executed at one exact point of simulation time which does not change during the execution. All parts of all processes' lives are merged together and arranged according to the value of their simulation time.

The execution of the simulation is divided into *steps*. One step corresponds to the execution of one selected process's part, having its own value of simulation time (not necesarrily unique). The execution is fully under the control of the currently executed process, i.e. no other process can interrupt or postpone the execution.

All processes share a *calendar* where *events* are stored. An event is an object holding information about a process's life part; this information contains the process's identification and the value of simulation time at which the life part is scheduled.

In order to divide their lives into parts, processes use *reactivation routines* which are able to establish reactivation points in the code of their lives. Two kinds of reactivation routines and reactivation points can be distinguished:

1. A *passivating routine* (*passivate*) terminates the current simulation step without adding any new event to the calendar; therefore, the process will not be activated anymore unless another process activates it explicitely.
2. A *temporarily passivating routine* (*hold(Δt)*) terminates the current simulation step and adds a new event to the calendar. This causes the process to be automatically restarted in the future, after Δt time units.

3 Design Decisions

The main goal of J-Sim was to provide a modern, easy-to-learn, and easy-to-use alternative to C-Sim (see [2] and [3]), existing since 1995. C-Sim is written in ANSI C and therefore is not object oriented. The user has to use some special constructs (macros) to define a process's life or an element of a queue and he must be aware of some unusual features, e.g. access to variables of a process. Process switching is implemented by using long jumps[1].

J-Sim uses Java threads for implementation of processes (one thread per process) and synchronization routines `wait()` and `notify()` to control their activity. The built-in support for concurrent programming was one of the most important arguments why Java was selected as the implemetation language, together with its platform independence and object orientation.

Most similar tools written in Java (see [5], [6], and [7]) are also based on Java threads, however, they are more complex and thus probably more difficult to learn.

4 J-Sim Core Classes

All J-Sim classes (except for GUI classes) are located in the package named `cz.zcu.fav.kiv.jsim`. It is necessary to import them at the beginning of every program using J-Sim. In this article, only the most important classes will be presented.

[1] `setjmp()` and `longjmp()` functions

4.1 The `JSimSimulation` Class

`JSimSimulation` instances represent theoretical simulation models where various number of processes and queues can be inserted. A calendar (instance of `JSimCalendar`) is owned by every simulation object, where events created by the simulation's processes are inserted. During one simulation step, exactly one event is interpreted and destroyed afterwards.

To the user, the simulation object offers the possibility of executing one simulation step by providing the `step()` method. During the execution, the thread calling this method (usually the main thread of the application) is suspended and it is reactivated as soon as the step finishes. Therefore, there is always one running thread only.

4.2 The `JSimProcess` Class

The `JSimProcess` class is a 'template' for user processes. A method, called `life()`, is introduced in `JSimProcess`, which contains the code representing behavior of a process. This method is initially empty and should be overwritten in user's subclasses.

There are four principal methods which can be used for process scheduling and switching: `passivate()`, `hold()`, `activate()`, and `cancel()`.

1. `passivate()` implements the passivating routine described above.
2. `hold()` implements the temporarily passivating routine described above.
3. The `activate()` method inserts a new event into the calendar and therefore assures that the calling process will get control in future. The method takes one parameter: the absolute simulation time of activation.
4. The `cancel()` method deletes all process's events from the calendar. If the process is passive, it will not be woken up anymore unless activated again by another process.

4.3 The `JSimHead` and `JSimLink` Classes

The `JSimHead` class is an equivalent of Simula's `HEAD`. It represents the head of a queue where objects of various types can be inserted. However, the class does not provide any methods for insertion or removal of data elements. Instead, the data to be inserted into a queue has to be wrapped by an instance of `JSimLink`, J-Sim's equivalent of `LINK`, which is able to insert/remove itself into/from a queue. `JSimHead`'s useful functions are: `empty()`, `cardinal()`, `first()`, `last()`, and `clear()` already known from Simula, and statistics functions `getLw()` and `getTw()`, returning the mean queue length L_W and the mean waiting time in queue T_W, respectively.

An instance of the `JSimLink` class can be inserted at most into one queue, using one of the following methods: `into()`, `follow()`, and `precede()`. The first one takes a queue as its argument while the others use another element, already present in a queue, to insert the caller into the same queue, either before or after the argument. A `JSimLink` object can remove itself from a queue by invoking its `out()` method.

5 Computing an Open Queueing Network Statistics with J-Sim

Let's show the possibilities of J-Sim on an example of a simple queueing network, depicted in figure 1.

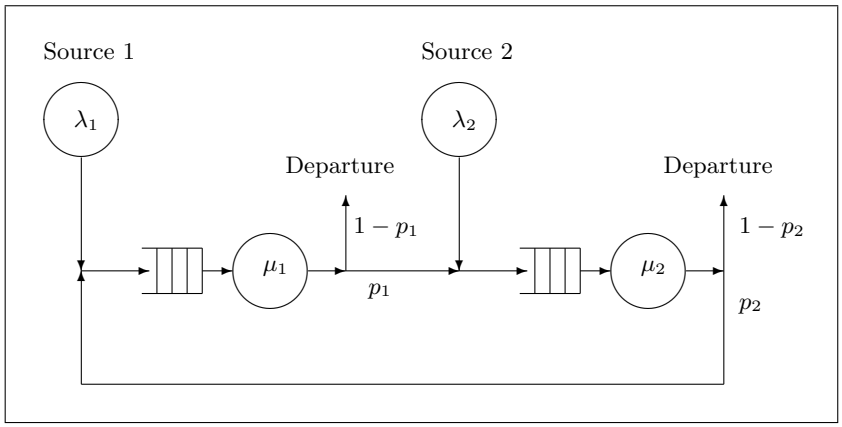

Fig. 1. An Open Queueing Network

The network contains two servers, each of them has a FIFO queue where transactions waiting to be serviced are put. The transactions (coming from two independent sources) enter the system at two input points and may leave it at two output points, after being served. We assume exponentially distributed random arrival time in the input streams of transactions and exponentially distributed random service time of both servers. The corresponding parameters of the simulation model are then as follows: λ_i is the mean frequency of the i^{th} input stream (and the parameter of the exponential distribution of arrival time), μ_i is the parameter of the exponential distribution of the i^{th} server's service time, p_i is the probability of transaction departure after being served in node i (and with complementary value $1 - p_i$, the transaction passes into node $3 - i$).

5.1 Theoretical Solution

Let's assume that the parameters of the network are set to the following values: $\mu_1 = \mu_2 = 1.0$, $\lambda_1 = \lambda_2 = 0.4$, and $p_1 = p_2 = 0.5$. To find the mean frequencies of the internal flow of both servers (Λ_i) under steady state conditions, we may use the model depicted in figure 2.

The two circles at the bottom stand for the servers, the two upper circles represent the outside environment where the transactions are generated and where they 'return' after being discarded. Obviously, $p_{0_A 1}$ and $p_{0_B 2}$ are equal

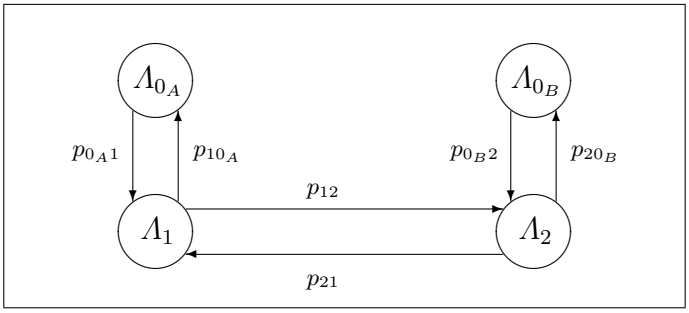

Fig. 2. Model of the Servers' Internal Flow

to 1 and Λ_{0_A} and Λ_{0_B} are equal to λ_1 and λ_2, respectively. The probabilities p_{ij} of transition from node i to node j can be expressed using the values from figure 1: $p_{12} = p_1$, $p_{21} = p_2$, $p_{10_A} = 1 - p_1$, and $p_{20_B} = 1 - p_2$.

Now we can proceed with finding Λ_i from the following system of linear equations:

$$\Lambda_1 = p_{0_A1} \cdot \Lambda_{0_A} + p_{21} \cdot \Lambda_2$$
$$\Lambda_2 = p_{0_B2} \cdot \Lambda_{0_B} + p_{12} \cdot \Lambda_1$$

After solving this system, we get: $\Lambda_1 = \Lambda_2 = 0.8$. Subsequently, we can compute the average load of both servers: $\rho_i = \frac{\Lambda_i}{\mu_i}$. Because $\rho_1 = \rho_2 = 0.8$, both servers are in steady state. Therefore, we can proceed with computing other characteristics for every server: the mean number of transactions waiting for service ($L_W = \frac{\rho^2}{1-\rho}$), the mean time in the queue ($T_W = \frac{L_W}{\Lambda}$), the mean number of transactions being served ($L_S = \frac{\Lambda}{\mu}$), the mean time of service ($T_S = \frac{1}{\mu}$), the mean number of transactions in the whole server ($L_Q = \frac{\rho}{1-\rho}$ or $L_Q = L_W + L_S$), and the mean response time ($T_Q = \frac{L_Q}{\Lambda}$ or $T_Q = T_W + T_S$).

Then, the mean number of transactions in the whole system can be computed as $L_Q = L_{Q1} + L_{Q2}$ and the mean response time as $T_Q = \frac{L_Q}{\Lambda_{0_A} + \Lambda_{0_B}}$. The results are shown in table 1.

Table 1. Theoretical Results – Characteristics of the Open Queueing Network from Figure 1

	Λ	ρ	L_W	T_W	L_S	T_S	L_Q	T_Q
Server 1	0.8	0.8	3.2	4.0	0.8	1.0	4.0	5.0
Server 2	0.8	0.8	3.2	4.0	0.8	1.0	4.0	5.0
Network	×	×	×	×	×	×	8.0	10.0

5.2 Solution Using J-Sim

We choose the classic approach to construct the model of the network – the servers and the sources of transactions will be active objects while the transactions will be passive.

Complete source texts can be found in directory `Examples/08_Queueing-Networks` of the distribution archive. (However, the values of the netwok's parameters have been changed.)

Transactions. A transaction is a simple passive object, holding no data except of the time of its creation. See file `Transaction.java` for details.

Sources of Transactions. Being an active object, the source has to be inherited from `JSimProcess`. It is assigned a queue where it stores the transactions generated during its life. See file `Generator.java` for complete source text.

```
public class Generator extends JSimProcess { // ...
protected void life() { // ...
 while (true) {
  link = new JSimLink(new Transaction(myParent.getCurrentTime()));
  link.into(queue);   if (queue.getServer().isIdle())
     queue.getServer().activate(myParent.getCurrentTime());
  hold(JSimSystem.negExp(lambda)); } /* ... */ }}}
```

Servers. Every server has a queue to take transactions from. If the queue is empty, the server passivates itself and it is restarted later when a transaction is inserted into its queue. After a transaction is taken out, the server processes it (simulated by `hold()`) and puts it into the other queue or throws away. The number of transactions (`counter`) and the time spent by them in the system (`transTq`) is registred for every server. Therefore, the mean response time of the whole network can be easily computed. See `Server.java` for more details.

```
public class Server extends JSimProcess { // ...
protected void life() { // ...
while (true) {
 if (queueIn.empty()) passivate();   else {
  hold(JSimSystem.negExp(mu));
  link = queueIn.first();
  if (JSimSystem.uniform(0.0, 1.0) > p) { // throw away
   t = (Transaction) link.getData(); counter++;
   transTq += myParent.getCurrentTime() - t.getCreationTime();
   link.out(); link = null; }
  else { /* insert again */ link.out(); link.into(queueOut);
   if (queueOut.getServer().isIdle())
    queueOut.getServer().activate(myParent.getCurrentTime());
   }}} /* ... */  }}
```

Running the Simulation. First, a simulation object has to be created. Then, two queues, two generators and two servers are created and the servers are assigned to the queues. Finally, the generators have to be activated. The servers are activated automatically as soon as a transaction is inserted into their empty input queue.

The simulation can be executed step-by-step when its `step()` method is repeatedly invoked, e.g. in a `while` cycle. Here, we let the simulation run until the simulation time reaches 1000 time units. Alternatively, we could use a `for` cycle and specify the number of steps to be executed.

```
while ((simulation.getCurrentTime() < 1000.0) &&
       (simulation.step() == true))     ;
```

Results. The program was run five times and the results shown in table 2 were obtained. The last two columns contain the average values obtained by the statistics functions `getLw()` and `getTw()` and the theoretical results from section 5.1.

Table 2. Results Obtained by the Program – Characteristics of the Open Queueing Network from Figure 1

	1	2	3	4	5	Average	Theoretical Result
L_{W1}	3.32	2.91	2.95	4.04	2.56	3.16	3.20
T_{W1}	4.27	3.70	3.70	4.92	3.38	3.99	4.00
L_{W2}	3.40	6.25	3.46	5.01	2.79	4.18	3.20
T_{W2}	4.38	7.52	4.21	6.25	3.74	5.22	4.00
T_Q	8.79	11.79	7.97	11.09	7.39	9.41	10.00

If the program is run more than five times or if we execute more simulation steps, we will probably get more accurate results, mainly concerning the queue no. 2.

6 Model of a Simple Parallel Algorithm

As another example, a model of a simple parallel algorithm executed at a shared-memory multiprocessor is presented. Several processes (with the same program) periodically utilize a block of shared data. A semaphore with conventional `P()` and `V()` operations is used to synchronize access to the data. The synchronized part of the program executed with all the processes is denoted as *critical section*.

The program of every process contains two main parts repeated in a loop: a block of local computation and a block where the shared data is updated inside the critical section. We are given two parameters concerning time conditions of the program: T_{out} (or $1/\lambda$, respectively) denotes the mean time spent by process outside the critical section and T_{CS} (or $1/\mu$) denotes the mean time

inside the critical section. We assume exponentially distributed random time spent by process inside each block.

Our goal is to find out the mean time of all processes' loops (taking into account the delays caused by the critical section) and the ratio between conflict-free-program frequency (as if there were no critical section) and the mean frequency of program with conflicts[2].

Let's say that there are two processes in the system ($N = 2$), $\lambda = 0.5$, and $\mu = 1.0$.

6.1 Theoretical Solution

First, we should construct a Markov-chain-based model of the system to compute the probabilities of different states of the system. The model is depicted in figure 3.

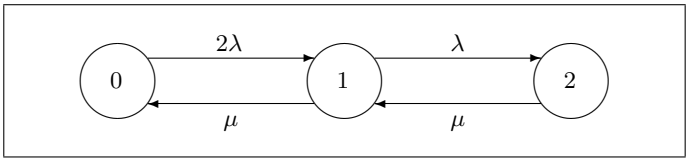

Fig. 3. Markov-Model of Two-Process System with a Critical Section

In state 0, both processes are computing locally. In state 1, one process is inside the critical section and the other one is outside. In state 2, one process is inside the critical section and the other one is blocked inside the `P()` operation of the semaphore guarding the critical section.

Since the system has no absorption states and the time spent in all states is exponentially distributed with parameter λ (or 2λ or μ), the following system of linear equations can be constructed:

$$\begin{aligned} 2\lambda \cdot p_0 &= \mu \cdot p_1 \\ \lambda \cdot p_1 + \mu \cdot p_1 &= 2\lambda \cdot p_0 + \mu \cdot p_2 \\ \mu \cdot p_2 &= \lambda \cdot p_1 \\ p_0 + p_1 + p_2 &= 1 \end{aligned}$$

Solving such system, we get: $p_0 = 0.4$, $p_1 = 0.4$, and $p_2 = 0.2$. Then, the frequency of loops (taking into account the possible conflicts) and its corresponding mean time can be evaluated as

$$f_{conf} = \frac{p_1 \cdot \mu + p_2 \cdot \mu}{2} = \frac{0.4 \cdot 1 + 0.2 \cdot 1}{2} = 0.3, \qquad T_{conf} = \frac{1}{f_{conf}} = \frac{1}{0.3} = 3.\bar{3}$$

[2] A conflict occures when a process invokes `P()` to enter the critical section but the critical section is already occupied by another process.

Without any delays caused by the critical section, f and T would be

$$f_{no\_conf} = \frac{1}{1/\lambda + 1/\mu} = \frac{1}{2+1} = 0.\bar{3}, \qquad T_{no\_conf} = \frac{1}{f_{no\_conf}} = \frac{1}{0.\bar{3}} = 3$$

Therefore, the performance decrease caused by the critical section is

$$Decrease = \frac{T_{conf}}{T_{no\_conf}} = \frac{3.\bar{3}}{3.0} = 1.\bar{1}$$

6.2 Solution Using J-Sim

Semaphores. Since data protection facilities, such as semaphores, are not necessary in non-concurrent environment, J-Sim does not offer them yet[3]. Therefore, we need to construct the semaphores as our own class. A semaphore has an integer counter (usually set to 1 at the beginning) and a queue where blocked processes are put.

```
public class Semaphore {
 private int counter; private JSimHead queue; // ...
```

The P operation decrements the counter if it is positive or blocks (passivates) the calling process and inserts it at the end of the queue if it is non-positive:

```
if (counter > 0) counter--; else { // ...
 link = new JSimLink(callingProcess); link.into(queue);
 callingProcess.passivate2(); } // else, ...
```

The V operation increments the counter if the queue is empty (no other process is entering the critical section) or takes a blocked process from the queue and restarts it. `V()`:

```
if (queue.empty()) counter++; else {
 firstLink = queue.first();
 firstProcess = (SemProcess) firstLink.getData();
 firstProcess.activate(myParent.getCurrentTime());
 firstLink.out(); firstLink = null; } // else, ...
```

Processes. Processes are active objects, therefore they have to be inherited from `JSimProcess`. They share an instance of `SharedData` (updated inside the critical section) and a semaphore – instance of `Semaphore`.

```
public class SemProcess extends JSimProcess { // ...
 private Semaphore sem; private SharedData data; // constr., ...
```

The `life()` method is very simple. An infinite `while` cycle contains both the 'local block' and the critical section protected by the semaphore. Real computation in every block is replaced with a call to `hold()` with an exponentially distributed random time as parameter. `life()`:

[3] They will be included in a future version of J-Sim.

```
while (true) {
 /* Local part */  myResult = JSimSystem.negExp(0.1);
 hold(JSimSystem.negExp(lambda));
 /* Critical section */ sem.P(this);
 data.setResult(0.99*data.getResult() + 0.01*myResult);
 data.incCountCS(); hold(JSimSystem.negExp(mu));
 sem.V(); } // while, ...
```

Results. The simulation was run five times, with simulation time limit of 30000 time units. The results are shown in table 3.

Table 3. Results Obtained by the Program – Characteristics of the Two-Process System with One Critical Section

	1	2	3	4	5	Average	Theor. Result
Mean number of loops	8951	9027	9061	9064	8966	9014	9000
Mean time of loops	3.3515	3.3233	3.3109	3.3098	3.3458	3.3283	$3.\bar{3}$
Mean freq. of loops	0.2984	0.3009	0.3020	0.3021	0.2989	0.3005	0.3
Performance decrease	1.1172	1.1078	1.1036	1.1033	1.1153	1.1094	$1.\bar{1}$

Conclusion

In this article, some basic facts about J-Sim have been presented, including its theoretical background. Being written in Java, a popular and easy-to-learn language, J-Sim should become at least as wide-spread as C-Sim, its predecessor. In the distribution package, there are included source texts, compiled classes, documentation and many examples. Today, J-Sim is a fully functional library which has been tested thoroughly, e.g. on the examples included in the package. J-Sim is available for free at [1].

References

1. J-Sim Home Page: `www.j-sim.zcu.cz`
2. C-Sim Home Page: `www.c-sim.zcu.cz`
3. Hlavička, J. - Racek, S. - Herout, P.: C-Sim v.4.1, Research Report DC-99-09, DCSE CTU Prague Publishing, Czech Republic, 1999
4. Holmevik, J.R.: The History of Simula, `java.sun.com/people/jag/SimulaHistory.html`
5. Desmo-J Home Page: `www.desmoj.de`
6. simjava Home Page: `www.dcs.ed.ac.uk/home/hase/simjava`
7. sim_tool Home Page: `monarc.web.cern.ch/MONARC/sim_tool`

Acknowledgement

This research was supported by the grant of the Ministry of Education of the Czech Republic, No. MSM-235200005 – *Information Systems and Technologies.*