University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

J-Sim – A Java-based Tool for Discrete Simulations

Jaroslav Kačer

Technical Report No. DCSE/TR-2001-05
September, 2001

Distribution: public

Technical Report No. DCSE/TR-2001-05
September 2001

J-Sim – A Java-based Tool for Discrete Simulations

Jaroslav Kačer

Abstract

This paper describes J-Sim v. 0.1.1, an object-oriented library using the method of pseudoparallel execution for simulation of discrete processes. Being written in Java, J-Sim is a fully portable successor to C-Sim, an already existing C-based library. The concepts used in both libraries are inherited from the Simula language. In this paper, the theoretical background as well as the basic principles of implementation in Java are presented.

This work was supported by the grant of the Czech Grant Agency "Developing SW Components for Distributed Environments", no. 201/99/0244.

Copies of this report are available on
`http://www.kiv.zcu.cz/publications/`
or by surface mail on request sent to the following address:

University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

1. Introduction

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

The execution of the simulation model is divided into *steps*. One step corresponds to the execution of one selected process' part. The simulation time changes discretely between two consequent simulation steps, according to the difference of simulation time between their corresponding life parts. The execution is fully under 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' life part; this information contains the process' identification and a value of simulation time at which the life part is scheduled.

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

1. A *passivating routine* terminates the current simulation step. It does not add any new event to the calendar; therefore, the process will not be activated anymore unless another process activates it explicitely. If it does so, this process will be re-run from the point where the routine was used. The name of the routine is *passivate*.
2. A *temporarily passivating routine* terminates the current simulation step and adds a new event to the calendar. This causes that the process will be automatically re-run in the future, after the amount of simulation time which is passes as the only argument of this routine. The name of the routine is *hold*.

2. Presentation of the Most Important Classes

All classes described below are parts of the package "`cz.zcu.fav.kiv.jsim`". It is necessary to import them at the beginning of every program using J-Sim. Although J-Sim consist of 21 classes put into 15 source files, only the most important of them will be presented here.

2.1 The JSimSimulation Class

`JSimSimulation` is a one of the fundamental J-Sim classes. Its instances represent theoretical simulation models where various number of processes and queues can be inserted. All processes and queues are registered and adopted by a simulation object upon their creation, they cannot be standalone. This is due to the simulation time which is shared by all processes and which is updated when a simulation step is executed.

A calendar (instance of `JSimCalendar`) is owned by every simulation object, where events created by simulation's processes are inserted. When the simulation object is asked by the user to execute one simulation step, the first event is taken from the head of the calendar and the simulation object tries to interpret it -- the simulation time must be changed and the process corresponding to the event must be activated and let running. After the interpretation is finished (the selected process returns control back to the simulation object), the event is taken out of the calendar and destroyed.

All processes present in the model have their description stored in a single-ended queue, called `infoQueue`. When a new process appears (is created by the user or by another process), the queue is updated, as well as in the case of a process' termination.

Two synchronization locks (instances of `Object`) are used: one for switching between processes and the simulation object -- `globalLock`, one for suspension of execution when the user asks to run the simulation in graphics mode and a graphics AWT window is open -- `graphicLock`.

To the user, the simulation object offers the possibility to execute one simulation step by providing the `step()` method. During the execution, the thread calling this method is suspended and it is reactivated when the step finishes. Therefore, there is always one running thread only, never more.

To use all possibilities of the Java language, the execution of simulations is not limited to the console mode only. Using the `runGUI()` method, the simulation object can redirect output and input to a graphics window which is automatically created. Then, the user can interactively influence simulation's progress by entering limit values of simulation time or number of steps to be executed.

2.2 The JSimProcess Class

The `JSimProcess` class is a "template" for user processes. Since it is a subclass of `java.lang.Thread`, its `run()` method's code has the possibility to be executed concurrently to other instances of this class. However, it is not this method which represents process' life. Instead, a new one, called `life()`, is introduced in `JSimProcess`. 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. The `passivate()` method suspends the calling process without creating a new event for it. If the process is not activated later by another process, it will stay passive forever. A process can invoke this method on itself only, not on another process. This method terminates the current simulation step, i.e. it separates different life parts of the process.
2. The `hold()` method temporarily suspends the calling process and reactivates it after the time specified as method's parameter. In fact, it adds a new event to the calendar and then passivates the process. Again, this method can be invoked on the calling process only and it terminates the current simulation step.
3. The `activate()` method inserts a new event into simulation's calendar. The new event belongs to the process whose method is invoked, not to the process which invokes it during its life. The method takes one parameter: absolute (simulation) time of activation. The method can be called by any process or even from outside of any process.
4. The `cancel()` method deletes all process' events from the calendar. If the process is passive, it will not be woken up anymore unless activated again by another process. A process can invoke this method on itself as well as on any other process.

A newly created process is automatically inserted into a simulation object's context which guarantees that the process will use simulation's shared lock for process switching -- it stores a reference to the lock upon its creation and synchronizes some pieces of code where switching takes place (`passivate()`, `hold()`) with this lock.

2.3 The JSimHead and JSimLink Classes

The `JSimHead` class is an equivalent of Simula's `HEAD`. It represents a 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 must be wrapped by an instance of `JSimLink`, JSim's equivalent of `LINK`, which is able to insert/remove itself into/from a queue. In fact, a `JSimHead` object just encapsulates a double-ended bidirectional queue of `JSimLink` elements, adding some useful functions:

1. The `empty()` method returns a logical value saying whether the queue is empty or not.
2. The `cardinal()` method returns the number of elements currently present in the queue.

3. The `first()` method returns the element being at the beginning of the queue. It returns a reference to `JSimLink`; therefore, the user must use other functions to get the real data.
4. The `last()` method returns the element being at the end of the queue. Again, not the real data but a reference to `JSimLink` is returned.
5. The `clear()` method removes all elements from the queue and sets queue statistics to their initial values. After being removed from the queue, the elements may be disposed automatically by JVM if there exists no other link to them.

The `JSimHead` class provides two statistics functions: `getLw()`, returning the average length of the queue, and `getTw()`, returning the average waiting time that elements have spent in the queue.

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 in the same queue, either before or after the argument.

3. Main Principles of Process Switching

The process switching in J-Sim is based on thread suspension and reactivation inside synchronized blocks of code. The principles are as follows:

1. A block of code is marked as synchronized. The synchronization uses an explicit lock which is shared by all processes and the simulation object.
2. Inside a synchronized block, the `wait()` method of the lock can be invoked. This causes the currently running thread to become passive.
3. Inside another block, synchronized with the same (shared) lock, the `notifyAll()` method of the lock is invoked. This causes all threads suspended using `wait()` (see point 2) to wake up and run. As soon as a woken-up thread leaves the synchronized block of code, another woken-up thread enters it (leaves the `wait()` method).

The switching takes place in the following methods:

1. In the `JSimSimulation.step()` method when the simulation object has selected a process to run in this step -- the simulation object must passivate the thread invoking `step()` and acivate the process.
2. In the `JSimProcess.getReady()` method when a newly created and started process must passivate itself not to execute commands that follow in the `run()`, respectively `life()`, method -- these commands can be executed only upon a request from the simulation object. No thread has to be activated here.
3. In the `JSimProcess.passivate()` method when the currently running process (the only running process) must passivate itself and activate the thread which invoked the `step()` method of the simulation object.
4. In the `JSimProcess.hold()` method. There is no difference between `passivate()` and `hold()`, concerning the switching.

Within the simulation model, every process is given an unique identification number -- `myNumber`. When the simulation object is asked to execute one simulation step, it stores the number of the selected process (process having an event at the beginning of the simulation's calendar) to the variable `runningProcessNum`, holding the identification number of the currently running process. Then, it send a signal to all processes which are passivated. When they are woken up, they compare the value of `runningProcessNum` with their identification number (`myNumber`) and either passivate themselves again or leave the synchronized block of code and continue running. Only one of them chooses the second possibility, the process intended to run.

When it decides to terminate the current step and the `passivate()` or `hold()` method is invoked, all threads, including the thread which invoked the `step()` method of the simulation object, are woken up by a signal from the currently running process. Before doing it, the process must set `runningProcessNum` to `NOBODY` which means that no process is intended to run. Therefore, all of them are passivated, only the thread invoking the `step()` method continues running and exits the method. The thread invoking the `step()` method is usually the main thread, i.e. the thread created automatically by JVM when it is given the name of a class having a method called `main()`.

When a simulation step is being executed, just one process is active -- the process selected by the simulation object. Outside any simulation step, all user processes are passivated in their `hold()` or `passivate()` methods.

4. Killing Processes

At the end of every simulation program, there is usually a number of processes passivated in their `hold()` and `passivate()` methods. To prevent the program from "hanging up" after the main thread finishes, they must be all destroyed before the end of the `main()` method is reached. A way how to do it is offered by the `JSimSimulation` class: the `shutdown()` method.

Every thread can be interrupted by another thread using its `interrupt()` method. When the thread to be interrupted is passivated in `wait()` and the interrupt signal is received, an instance of `InterruptedException` is thrown out from the method. In J-Sim, this exception is caught immediately and instead of it, a new exception, instance of `JSimProcessDeath`, is thrown out. `JSimProcessDeath` is a subclass of `RuntimeException` and thus need not be caught explicitely.

After being thrown out, it is propagated to an upper level, which is usually the `life()` method. Being not caught here (the `life()` method is programmed by the user), it continues to the `run()` method where it is finally caught in an exception handler. After that, the `finish()` method is called, which deregistres the process from the simulation model in the usual way.

A little summary of `JSimProcessDeath`'s spreading follows. It contains a list of all possible paths through which this exception can spread:

`getReady()` > exception handler in `run()`
`hold()` > `life()` > exception handler in `run()`
`passivate()` > `life()` > exception handler in `run()`

It should be noted that processes can also terminate their life by reaching the end of the `life()` method, as it is usual for finite processes. Then, no exception is thrown out, after leaving the `life()` method, the `finish()` method is called, deregistering the process from the simulation model, and the process dies right after that by reaching the end of `run()`.

5. An Example of Use

A little piece of code will be shown in this chapter to show the ease of using J-Sim. First, a new class (subclass of `JSimProcess`) is created, named `PeriodicProcess`. Instances of this class periodically print out a character which is passed to their constructor as parameter. Note that only the constructor and the `life()` method are rewritten, nothing more. Parts of their life are divided by the `hold()` method whose parameter is a randomly generated number having exponential distribution. The class must be put in a file called `PeriodicProcess.java` and then compiled.

```
import cz.zcu.fav.kiv.jsim.*;
public class PeriodicProcess extends JSimProcess
{
 private char           charToPrint;
 public PeriodicProcess(JSimSimulation simulation, char c, String name) throws
JSimSimulationAlreadyTerminatedException, JSimInvalidParametersException,
JSimTooManyProcessesException
 {
  super(name, simulation);
  charToPrint = c;
 } // constructor

 protected void life()
 {
  try
  {
   double time;
   while (true)
   {
    time = myParent.getCurrentTime();
    message(getProcessName() + " at " + time + " : " + charToPrint);
    hold(JSimSystem.negExp(1.0));
   } // while
  } // try
  catch (JSimSecurityException e)
  {
   e.printStackTrace();
   e.printComment(System.err);
  }
 } // life
} // class PeriodicProcess
```

In the main program, two processes are created and activated. Then, the `step()` method of the simulation object is invoked repeatedly until the simulation time reaches or overpasses 10.0. The code must be placed in a text file which is named `SimpleSwitchingExample.java`, compiled and then run.

```
import cz.zcu.fav.kiv.jsim.*;
public class SimpleSwitchingExample
{
 public static void main(String[] args)
 {
  PeriodicProcess       a, b;
  JSimSimulation        simulation = null;

  try
  {
   simulation = new JSimSimulation("Simple Switching Example");
   a = new PeriodicProcess(simulation, 'A', "Process A");
   b = new PeriodicProcess(simulation, 'B', "Process B");
   a.activate(0.02);
   b.activate(0.03);

   while ((simulation.getCurrentTime() < 10.0) && (simulation.step() == true))
    ;
  } // try
  catch (JSimException exc)
  {
   exc.printStackTrace();
   exc.printComment(System.err);
  }
  finally
  {
   simulation.shutdown();
  }
 } // main
} // class SimpleSwitchingExample
```

Conclusions

In this article, some basic facts about J-Sim have been presented, including the theoretical background on which it is based. Being written in Java, a popular and easy-to-learn language, J-Sim should become at least as spread as C-Sim, its predecessor. Since J-Sim and C-Sim are tightly related to each other, a possible migration to J-Sim should be a matter of hours. In the distribution package, source texts, compiled classes, detailed API documentation and many ready-to-run examples are included. Today, J-Sim is a fully functional library which has been tested on the examples included in the package.
J-Sim is available for free at `http://home.zcu.cz/~jkacer/jsim`.

References

[1] *Kačer, J.*: **J-Sim -- A Java-based Tool for Discrete Simulations**, Diploma Thesis, May 2001, University of West Bohemia, Pilsen, Czech Republic
[2] `http://home.zcu.cz/~jkacer/jsim`
[3] `http://www.c-sim.zcu.cz`

Acknowledgement

This research was supported by the grant of the Czech Grant Agency *"Developing SW Components for Distributed Environments"*, no. 201/99/0244.