University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering

J-Sim

A Java-based Tool for Discrete Simulations

Jaroslav KAČER

May 2001

Contents

List of Figures

J-Sim version 0.1.1

Jaroslav KAČER
jkacer@students.zcu.cz
University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering
Pilsen, Czech Republic

Abstract

J-Sim is a Java-based tool for simulation of discrete processes. J-Sim was developed at the University of West Bohemia in order to give programmers a possibility to choose between two of the most used programming languages in these days: the C language and Java. In recent years, Java has gained a stable position in programmers' community and has become widely supported by all subjects in the IT market. Thus there is a strong belief that J-Sim will be used at least as much as C-Sim, an already existing C-based simulation library by which J-Sim was strongly inspired.

Both C-Sim and J-Sim have their roots in Simula, a programming language created especially for simulation purposes. However, J-Sim provides a modern, more elegant, comprehensible, and accessible way to do the same job.

The typical application area of J-Sim is functional validation of distributed, parallel, and fault-tolerant systems and programs.

J-Sim is freeware, available for download at its WWW page. Source texts, API documentation, and many examples are also provided.

Keywords: Discrete-event simulation, Pseudo-parallel processes, J-Sim, Java, C-Sim, Simula.

1 Introduction

Before going deeply into details of the J-Sim system, a presentation of the theoretical background on which J-Sim and other similar tools are based should be given. This chapter is especially important for those who have never modelled any simulation of discrete processes, but can be useful for experienced users too, because it clarifies and explains all principles used in discrete simulations, and presents some recommendations how such a discrete-simulations modelling tool should work.

1.1 Fundamental Concepts of Discrete Simulation

As the word "discrete" says, *discrete simulation* is not executed continuously. Instead, it is divided into *steps*. Every simulation has its own time which will be called here *simulation time*. A simulation can be either finite or infinite. Wheter a simulation is finite or not cannot be determined easily because new steps can be created dynamically during execution of another step. Each step is executed at one exact point of simulation time. This time cannot be changed in any way during one simulation step. Two or even more steps can be executed at the same simulation time but it is not determined which of them will run first. Steps are never executed parallelly, but always sequentially. The time difference between two consequent simulation steps is usually a non-zero positive value. When one simulation step has finished and another one is being started, the simulation time must change – it jumps from the old value to the new.

In fact, a simulation is just a passive object which does not do anything. But it can contain various number of processes which are active. The number of processes within a simulation can vary as time passes, existing processes can die, and thus disappear completely, and new processes can be created. Processes are active thanks to their pre-programmed *life*. Usually, processes do not share any common data but if there is such a need, it can be easily arranged. The thing which gives processes the possibility to be executed at several different time points is called *reactivation points*. A reactivation point is such a point in a process' life which separates parts to be executed during different simulation steps, and therefore at different times. This will be discussed in more detail in a following chapter.

However, the time differences between two consequent parts of a process' life does not necessarily correspond to the differences between two consequent simulation steps. The fact which breaks up this equality is that one simulation contains several processes, all of which have their corresponding life parts to be executed at different time points. Therefore, the simulation

must always evaluate time at which the next step will be executed as the minimum value of all values requested by all processes within the simulation. The processes' life parts are *interleaved* – there can be number of simulation steps to be executed between two parts A and B of the same process. This number can be even unknown when the life part B is scheduled because another process run in the time interval between A and B can create thousands of life parts with differences between them so small that they will fit into the interval $< A, B >$.

To be able to work in the manner described above, every simulation has its own *calendar* where *events* are stored. An event is an object holding information about a required time point and a process that wants to be run at that time. All of the events are stored in the calendar in ascending order with regard to the time of events. When a new event is added to the calendar, it is inserted at a corresponding position, i.e. the time of previous event in the calendar is less than or equal to the new event's time, and the time of next event is greater than or equal to it.

An example on what it may look like is shown in the following figure:

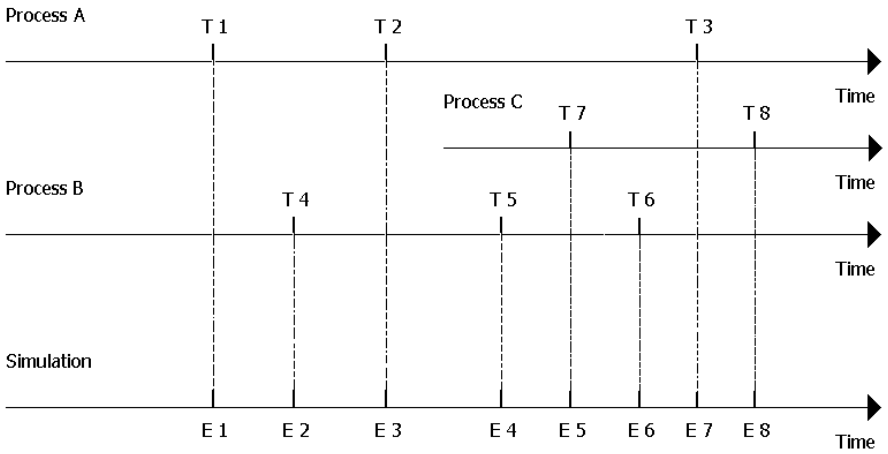

Figure 1: The Principle of Event Interleaving

Two processes (A and B) want to be executed; each of them has three

events to be scheduled. Process A's events are scheduled at time points T_1, T_2 and T_3, process B's events at time points T_4, T_5 and T_6. Because T_4 is greater than T_1 but less than T_2, it is put between these two events, and so on. Then, a new process, called C, is created and its' two events are inserted into the calendar. Although process A's and process B's events T_3 and T_5 (and perhaps T_6) are already present in the calendar, process C's event T_7 is inserted between them and its corresponding simulation step (a piece of process C's life) will be executed after T_5 and before T_6.

Note that none of the three processes knows anything about the two other processes' events. It does not even know that others may exist! It is a matter of the simulation to put all the required events in a good order and to select an appropriate process to run when it is demanded to execute a simulation step. Also note that eight simulation steps are needed to *consume* all the events depicted in the figure. But new events can be created during execution of the eight ones shown. This is the principle of *infinite simulations*. A simulation is infinite if it contains at least one *infinite process*. A process is infinite if it never reaches the end of its life, i.e. for every consumed event there is on average at least one new event created. The easiest way how to make a process infinite is to use a neverending loop which will contain a *reactivation point* in its body.

1.2 Simulation-Time to Real-Time Mapping

So far the simulation time and its property of discreteness has only been discussed. However, when a program is executed it is a continuous process of taking instructions and interpreting them. In J-Sim, there exists an exact way how to describe the relation between the *simulation time* and the *real time*. To better understand this subject, look at the scheme 2 on page 4.

There are four events to be executed, each of them at a different time. Between them there are *time gaps* which can have any arbitrary length. The length can be zero, but that is a very special case discussed above – more simulation steps scheduled at the same time. The most important thing to be seen on the bottom axis is that every simulation step is zero time units long. That is a consequence of the fact that time cannot change during a simulation step.

At the upper axis which represents real time, a corresponding time interval for each of the four simulation steps can be found. They have a non-zero length, i.e. the execution of a step in real life is never infinitely short. Note that not all the time is spent by executing a process' code. There are little time intervals necessary for the system to switch between processes and there can be also time intervals that have nothing to do with the simulation at all.

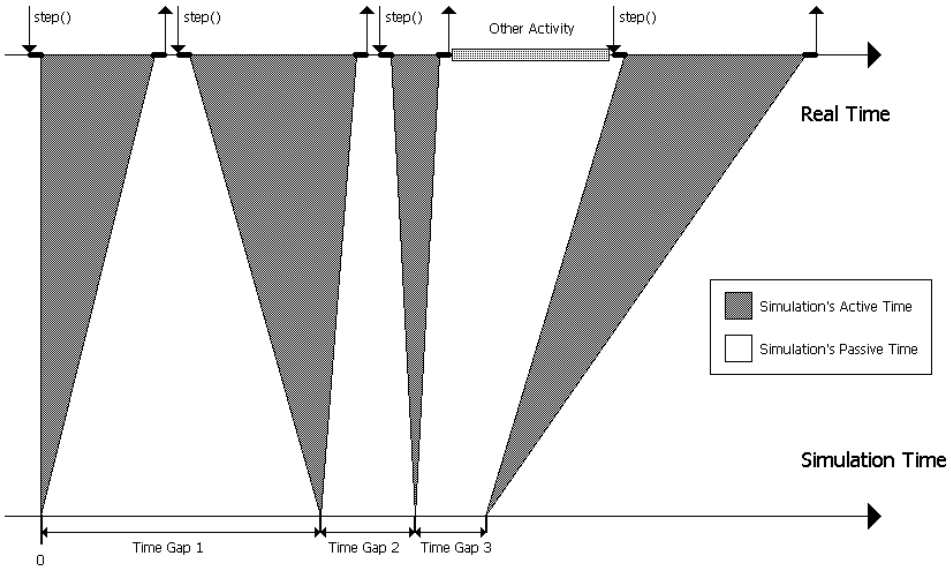

Figure 2: Simulation-Time to Real-Time Mapping

This interval is labeled as "*Other Activity*".

A simulation step begins when the user asks the simulation to execute a step. The simulation must select an appropriate process[1] and give it control. In that moment, the process gains full control over its life and nothing (neither the simulation nor other processes) can stop or interrupt it. It is fully in its own competence when the step will be finished. There are two ways how to finish a step. The process can either naturally die (i.e. its life terminates) or it can use a *reactivation routine* to establish a reactivation point. In the latter case, the process will be temporarily interrupted until another event belonging to this process is found in the calendar. Such an event can be added to the calendar either by the reactivation routine itself or by another process. In the former case, the process will not be able to run anymore.

1.3 Reactivation Points and Reactivation Routines

A reactivation point is such a point in a process' life where the process is temporarily interrupted (or suspended) until an event belonging to the process

[1]It is a process which has an event at the top of simulation's calendar.

is found in the simulation's calendar. Every reactivation point terminates currently executed simulation step. As stated above, the currently running process has absolute power over simulation's real time. It can even deprive all other processes of the possibility to run by performing a neverending loop which does not contain any reactivation points. Once such a process is run and gets to this point, no other process will have a chance to run. Plus, the simulation step during which this process is run will never terminate and the programmer will therefore loose control over simulation's progress. Fortunately, such cases are completely useless in the domain of discrete simulations.

Every reactivation point is realized by a reactivation routine. In general, two kinds of reactivation points and their corresponding reactivation routines can be distinguished:

- A *passivating* routine terminates current simulation step and returns control to the simulation. It does not add any new event to the simulation's calendar. Therefore, the process which uses this routine will not be activated anymore unless other processes activate it. The process cannot be sure if and when it will be activated. The name of this routine is *passivate*.

- A *temporarily passivating* routine terminates current simulation step and returns control to the simulation, but in addition creates a new event belonging to this process and inserts it into the simulation's calendar. Therefore, the process can be sure that it will be activated in the future. This routine has one parameter determining the difference between current time and the time when the process has to be reactivated. The name of this routine is *hold*.

1.4 Scheduling Possibilities

In order to have a complete and more detailed overview of existing possibilities of scheduling, the following paragraphs will briefly characterize them.

First of all, every process should be activated to have a chance to run at all. The name of this operation is *activate* and it has one parameter determing the time of activation. A process can activate other processes at any time but the activation is usually done by the main program before the simulation starts. Once a process is activated and run it does not need another activation to run at another time, it can use the temporarily passivating routine.

Sometimes there is a need to cancel an already made scheduling and delete all events belonging to a process from the calendar. The name of this

operation is *cancel*. As a result, the affected process will not run anymore unless it is newly activated. A process can cancel another process as well as itself.

The two remaining possibilities have been already mentioned above. It should be noted that the *passivate* operation does not alter the simulation's calendar in any way while the *hold* operation does. A process cannot passivate or temporarily passivate another process, only itself. Every attempt to passivate another process will result in an error.

1.5 Requirements for the Tool

Having the theoretical knowledge obtained in previous chapters, one can try to formulate some necessary requirements that the simulation tool should meet. These requirements concern the tool itself, but also the programming language in which the tool will be written:

1. The programming language should be widely spread and well known to most developers (scientists).

2. The programming language should be *platform-independent* in order to allow users to use the tool in all possible computing environments and to port user's simulations from one platform to another with minimum effort. *Complete portability* (source texts and runnable output) is preferrable to *partial portability* (source texts only, need to recompile in target environment).

3. The programming language should offer a way how to create and run several processes concurrently, providing a safe method of their *synchronization* – temporary suspension followed by reactivation upon a received signal is necessary.

4. The programming language should be object-oriented. This will allow the tool to provide basic classes that could be extended both in functionality and in data content by users.

5. The tool should offer an easy way to create a simulation and a process. The preferrable way is the inheritance from already existing classes present in the tool.

6. The tool should hide all implementation details and make process suspension and reactivation transparent for the user.

7. The tool should provide two ways of running: in batch mode with output to the console, and in interactive mode with output to a graphics window. Therefore, graphics possibilities should be offered by the programming language and they should be preferrably platform-independent as well. In the latter case, the simulation should be driven by user's actions.

Now each requirement will be taken point-by-point to determine whether J-Sim meets it or not, and if it does, then in what way:

1. J-Sim is written in 100%-pure Java. Today, Java is one of the most spread and well-known languages. Its runtime environments and compilers are available for free for most widely used platforms. Java is easy to learn and easy to use.

2. Java pre-compiled code is interpreted in the target environment; therefore, both source texts and pre-compiled code are portable. In case of any problems, source texts provided with J-Sim can be used to generate new code, compiled in the target environment, thus 100-percent compatible with JVM[2] used.

3. Java provides a class called `Thread` whose instances run parallelly with other such instances. Thread support is directly built into the language and therefore no additional library is necessary, like in the case of the C language. An efficient way of thread synchronization is provided directly in the language too. Every object has its own lock which can temporarily suspend currently running thread and reactivate it when a wake-up signal is received from another thread.

4. Java is a fully object-oriented language, providing the concepts of classes, instances, encapsulation, inheritance and polymorphism[3]. Unlike in C++, the use of object principles is strictly mandatory in Java.

5. J-Sim provides basic classes for simulation, process and queue. These classes can be either directly used or extended according to specific user's requirements.

6. There are no special actions required in order to passivate or temporarily passivate a process. Two methods are provided in the process class whose use is very intuitive and easy. The user need not know any implementation details concerning suspension and reactivation.

[2]Java Virtual Machine

[3]If you do not know what these terms mean, consult a textbook of object-oriented programming.

7. J-Sim provides two possibilities of running a simulation which can be dynamically switched. The first one is the *batch mode*, sending output to the console. The second one is the *interactive mode*, using a graphics window to control the simulation and to display simulation output. Both of them use only standard Java services and thus are fully portable. However, the possibilities of the target environment may limit their use[4].

To avoid confusion at this point, it should be defined what exactly J-Sim is and what it is not. J-Sim is a tool which can help users to simulate behaviour of pseudo-parallel processes. J-Sim is not a complete, ready-to-run software, it always needs a *user* (programmer) who creates a simulation and processes and pre-programs them a meaningful sequence of actions called *life*.

[4]For example, older versions of MacOS do not have console output.

2 Inside J-Sim Kernel

In this chapter, an explaination of the basic principles on which the J-Sim kernel is based will be presented. It will discuss mainly the way in which processes are switched, how they are suspended and reactivated and the relationship between a simulation and its processes. It is assumed that the reader has already some knowledge of the Java language and concurrent programming.

2.1 Possibilities Offered by Java

The Java programming language provides a set of classes, interfaces, methods and keywords manipulating with threads. Only some of them will be mentioned, those which are particularly interesting for realization of pseudo-parallel execution.

2.1.1 The `Thread` Class

First of all, the `Thread` class must be mentioned. An instance of this class or its subclass[5] is able to be executed parallelly with the thread that created and ran it. Every instance of this class has its parallelly executable code stored in the `run()` method which is empty in the case of `Thread`. This method should be always overwritten in order for the thread to provide some meaningful concurrently-run services. When an instance of a `Thread`'s subclass is created, it is passive like all ordinary Java objects. It is switched to an active state using its `start()` method. This causes the code of its `run()` method to start to be executed parallelly. Inside this method, other methods (of the same object as well of other objects) can be called without breaking up the parallelism. Once the `run()` method reaches its end, the object becomes passive again without the possibility to be restarted. But it can still provide non-parallel services by methods invoked in the usual way. There is always at least one thread within every Java program: the `main()` method of the class which was given as a parameter to the Java interpreter. Here is a little example how to create and run a thread which prints out a message:

```
class MyThread extends Thread
{ public void run()
  { System.out.println("Hello."); }
}
```

[5]Subclass is a class inherited from the main class using the `extends` keyword.

```
. . .

// In main()
MyThread thread = new MyThread();
thread.start();
```

2.1.2 Unusable Synchronization Methods

The `Thread` class provides three very promising methods allowing to manipulate with a thread's execution state: `suspend()`, `resume()`, and `stop()`. The `suspend()` method temporarily suspends execution of a thread until `resume()` is invoked on the same thread. The last method interrupts execution of a thread and causes it to exit its `run()` method. However, all of them have been marked as *deprecated* since JDK[6] 1.2. There were very serious reasons to do so, mainly concerning security. The reasons can be found in [Lea].

2.1.3 Usable Synchronization Methods

However, Java provides a very efficient way of *thread synchronization*, although it is not obvious for the first sight. A method[7] of a class can be marked as synchronized (using the `synchronized` keyword in its header) which will prevent all threads from entering method's body if there is already another thread. A *lock* is used to keep information whether it is possible to enter the method. If there are more protected methods within a class, they share the same lock. Note that the lock belongs to an object (instance of a class) not to a class. Here is a simple example of two synchronized methods sharing the same lock:

```
public synchronized void a()
{ /* ... something is happening here ... */ }

public synchronized void b()
{ /* ... and something else here ... */ }
```

When a thread enters the `a()` method, the lock is acquired and no other thread can enter this nor the `b()` method. All threads attempting to do so are temporarily suspended and one of them is resumed when the first thread leaves the `a()` method and the lock is released.

[6] Java Development Kit

[7] Here, according to standard OOP terminology, "method" stands for a function of a class, not a method of synchronization.

However, methods or just their parts can be synchronized by another lock than the implicit one (`this`). The following piece of code protects the methods' code in exactly the same way as the last one, but uses an explicit lock which must have been properly initialized. The lock can be instance of any class.

```
public void a()
{ synchronized (theLock)
  { /* ... something is happening here ... */ }
}

public void b()
{ synchronized (theLock)
  { /* ... and something else here ... */ }
}
```

Note that exactly the same method of thread synchronization is used in J-Sim. In order to be able to synchronize their methods with methods of all other processes, processes must share the same lock. This will be described in the following chapters.

There are techniques offered by Java that allow a thread to become temporarily suspended inside a synchronized method (or just a block of code) in order to release the lock and thus allow another thread to enter the synchronized method or block. Method `wait()`, introduced already in the class `Object`, provides this service. The suspended thread must be woken up sometimes not to spend all its life in passive state, being not able to do anything. There exist two methods usable for this purpose: `notify()` and `notifyAll()`. They don't differ in behavior if there is just one thread suspended. If there is more than one thread suspended, the `notify()` method wakes up just one of them, it is not specified which one. The `notifyAll()` method wakes up all of them. However, they cannot start to run parallelly since they wake up inside a synchronized block of code. Only one of them is allowed to run (not specified which one) and all others must wait until the selected thread leaves the synchronized block of code. As soon as it does so, another already woken-up thread is selected and allowed to run and so on. A typical example on how to use these methods (`notify()` is omitted here but the principle is the same) is as follows:

```
public void sleepUntilWokenUp()
{
 synchronized (theLock)
```

```
 {
  // This piece of code will be executed right after entering
  // the synchronized block. This can take a while, too,
  // if there is already another thread.
  theLock.wait();  // suspended
  // This piece of code will be executed
  // after the suspended thread
  // is woken up and the thread that woke him
  // up in wakeThemUp()
  // leaves the synchronized block.
 }
}


public void wakeThemUp()
{
 synchronized (theLock)
 {
  // This piece of code will be executed right after entering
  // the synchronized block. This can take a while, too,
  // if there is already another thread.
  theLock.notifyAll();  // wake up all suspended threads
  // This piece of code will be executed right after waking other
  // processes up, there is no time delay.
 }
}
```

Classes having these synchronized methods are not usually threads but *passive data containers*. They use synchronization to protect their data from being overwritten or read in a bad way by two concurrently running threads trying to change and read the value of a variable in the same moment. Such classes are called *monitors*.

However, nothing prevents programmers from implementing this mechanism into a class which is a subclass of `Thread`. But in this case, the programmer must be extremly careful about *liveness* od such threads. It is obvious that they cannot use their private locks. If a thread passivated itself, it would not be woken up anymore since other threads, trying to wake it up, would use their own private locks (`myPrivateLock.notifyAll()`). Therefore, the waking-up signal would never reach the suspended thread! Instead, all threads must use the same lock. Then, it is sufficient if at least one thread is in runnable state because it is able to wake up all others by invoking the

`notifyAll()` method on their common lock. Exactly the same principle is used by J-Sim kernel to switch between user processes and the main thread, requesting to do one simulation step.

2.2 Simulation as Container

As it has been mentioned above, in order for user processes to switch properly they need to have one unique lock shared by all of them. Trying to analyze the situation deeper, one will find that there is not running the same number of threads as the user processes. There is always one in addition to them – the main thread, which executes the code written in method `main()` of the class whose name was given as the first parameter to Java interpreter `java` (or `java.exe`). This thread is created and run automatically by JVM. It is usually this thread which calls the `step()` method of the simulation but does not necessarily need to be. Because it is required that this main thread wait until the step is finished (and all user processes are suspended), it must be synchronized by the same lock as processes are. Actually, it is not the main thread which owns the lock. The *lock is owned by the simulation* and a reference to it can be obtained by invoking simulation's method `getGlobalLock()`. In fact, the main thread need not care about this lock at all. When it invokes the `step()` method of the simulation, it has full access to all its variables, therefore to the lock too.

The situation is depicted in figure 3. The lock named `globalLock` is owned exclusively by the simulation object and its processes get just a reference to it. Because every process has its simulation to which it belongs (called `myParent`) it can get the reference upon creation (in constructor `JSimProcess`) by invoking `myParent.getGlobalLock()`. The fact that the reference's name is `ourCommonLock` does not have any influence. Despite different names in different classes it is still the same object. Now, all user processes and any arbitrary thread invoking the `step()` method of their simulation can be synchronized by the same lock.

Ignoring the rest of the image for the moment, the process switching using this lock will now be examined.

2.3 Execution of a Step – Switching Techniques

2.3.1 Points of Switching

Before explaining the switching techniques it would be convenient to state where exactly the switching takes place. In class `JSimSimulation`, it is:

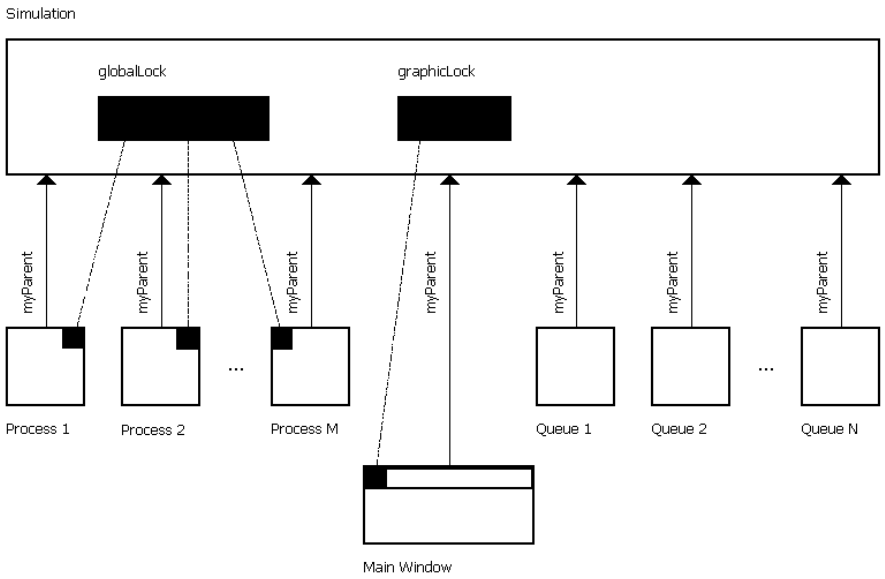

Figure 3: Simulation as Container

- in the `step()` method when the simulation has selected a process to be run in this step – the simulation must passivate the main thread and activate the process.

In class `JSimProcess`, switching points are:

- in the `getReady()` method when a newly created and started process must passivate itself not to execute commands that follow – these commands can be executed only upon a request from the simulation. This cannot be considered as full switching since no thread has to be activated because the simulation (the thread calling the `step()` function, respectively) has not been suspended;
- in the `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. This is the full switching since there is only one thread running before and one after the switching point and they are not identical;
- in the `hold()` 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. The technique of switching does not differ at all from the last case, the `hold()` method just creates a new event in simulation's calendar. Again, this is the full switching.

2.3.2 Commands Used to Switch

The deterministic switching[8] is possible thanks to the following factors:

- A common lock, owned by the simulation, exists. This lock is called `globalLock` in `JSimSimulation` and `ourCommonLock` in `JSimProcess`.
- A calendar of events, owned by the simulation, exists and processes have right to add new events into the calendar. This calendar is called `calendar`.
- Every process is identified by a unique number within its simulation (`myNumber`) and the simulation has a variable determining the number of currently running process: `runningProcessNum`. There exists a special constant `NOBODY` saying that no process should run, only the main thread. Processes cannot change the value of this variable; this operation is reserved exclusively for the simulation. However, they can use simulation's `switchToNobody()` method which sets `runningProcessNum` to `NOBODY`. This means that a process cannot switch to another process, only to the main thread. This guarantees that during one call to `step()` there will be executed really one step only, not more.

Let's have a look now how the switching is implemented in Java. First, switching from the main thread to a process (a part of `JSimSimulation.step()`):

```
runningProcessNum = calendar.getFirstProcessNum();
time = calendar.getFirstProcessTime();

/* Now sending the wake-up signal to all threads in the system */
globalLock.notifyAll();

/* Now waiting until "hold()" or "passivate()" is called. */
while (runningProcessNum != JSimCalendar.NOBODY)
{
```

[8]Such a kind of switching where it is determined in advance which process will run as next.

```
 try
 {
  globalLock.wait();
 }
 catch (InterruptedException e)
 {
  /* Just ignoring this exception -- nobody
     should iterrupt the main thread */
 }
} // while
```

First, the `runningProcessNum` variable is set to the value which is taken from the top of simulation's calendar. This process is intended to be run in next step. Then, the time of the simulation is set to the value of the first event in the calendar. After that, all processes belonging to the simulation are woken up by a signal sent by `notifyAll()`. When they receive the signal, they check whether they are intended to run (they compare their number, `myNumber`, with the result of the call to `myParent.getRunningProcessNum()`). If a process finds that it is its turn, it just continues. All others suspend themselves again.

After notifying all processes, the main thread passivates itself, using `globalLock.wait()`. Note that the notification must be done before passivation – a passivated process cannot notify any other process since it is stuck at one point. The `wait()` method may throw out an `InterruptedException` if the thread is interrupted by another thread in suspended state. Here, the case that such an exception is thrown out is simply ignored because it is not expected here. When the main thread is woken up, it checks whether it is intended to run (`runningProcessNum` is set to `NOBODY`) and continues running or passivates itself again. Because processes may only switch back to the main thread, this condition will always be `false` and the cycle will never run twice. However, it is a widely accepted convention to wrap every `wait()` into a `while` cycle. It is completely harmless and illustrates well the programmer's intentions.

Now, a look at how processes switch back to the main thread will be given. The method of switching in `hold()` is identical to the method in `passivate()`, thus it will be explained at once:

```
synchronized(ourCommonLock)
{
 /* Wake up main() which is waiting now in step() */
 myParent.switchToNobody();
```

```
 ourCommonLock.notifyAll();

 /* Going to sleep */
 while (myParent.getRunningProcessNum() != getProcessNumber())
 {
  try
  {
   setProcessState(STATE_PASSIVE);
   noOfCalendarEntries--;
   ourCommonLock.wait();
  }
  catch (InterruptedException e)
  {
   shouldTerminate = true;
   requestsToTerminate++;
   break;
  }
 } // while
 setProcessState(STATE_ACTIVE);

 /* If we have to die this exception is propagated till run() */
 if (shouldTerminate)
  throw new JSimProcessDeath();
} // synchronized
```

This is a more complicated example of switching since it must be able to hanle process interruptions. Processes are interrupted by the simulation when the user wants to terminate the program and calls the `shutdown()` method. The way a process can be terminated is subject of the next chapter.

First, the process tells the simulation that it wants to passivate itself and the main thread should run – `myParent.switchToNobody()`. Then, the waking-up signal is sent to all threads passivated with the same lock. After checking the number of process intended to run, they all passivate themselves again, except for the main thread which invoked the `step()` method of the simulation. Then, the currently running thread passivates itself (`ourCommonLock.wait()`). When it is woken up, in the next simulation step when the simulation needs to switch to the selected thread, it compares the number of the process selected to run (which has changed since this process' passivation) to its own number and decides whether to run or passivate itself again.

If a process is interrupted by the mechanism mentioned above and discussed in the next chapter, it sets its `shouldTerminate` to `true` and throws out a new instance of `JSimProcessDeath` exception.

2.3.3 An Example of One Simulation Step

To better illustrate what has been said about process switching, figure 4 on page 18 gives an explanation.

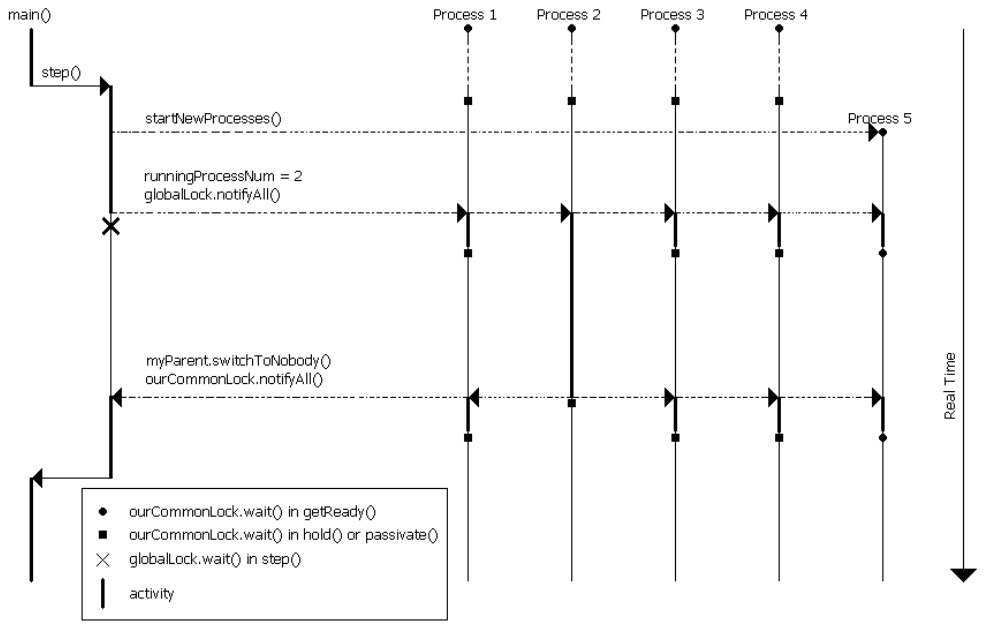

Figure 4: An Example of One Simulation Step

When the main thread asks the simulation to execute a step, there are four processes already present within the simulation. They are all suspended in the `passivate()` or `hold()` methods. Some of them could be suspended in the `getReady()` method, too, if they have not run yet. Because J-Sim allows processes to be created dynamically when the simulation is already running, at the beginning of every step the newly added processes must be started. This task is performed by `startNewProcesses()`. In this example, one process was added to the simulation in the last step and must be started now. As soon as it is started, it passivates itself in the `getReady()` method.

After new processes are started, the simulation looks into the calendar of events (using `calendar.getFirstProcessNumber()`) and finds that it is

process number 2 which should be run in this step. It stores this number into the variable `runningProcessNum` and sends a wake-up signal to all suspended processes. Immediately after that, the currently running thread is passivated.

When processes (including the newly created one) receive the signal and are woken up, they must decide whether to run or to passivate themselves. Most of them choose the second possibility. Only one of them, the process having number 2, decides to continue. It continues running a part of its life as the only thread within the simulation until it calls the `hold()` or `passivate()` method.

When it does so, `runningProcessNum` is set to `NOBODY` by calling `myParent.switchToNobody()`. Then, all threads are woken up again by `notifyAll()`, including the main thread. Immediately after notifying them, the process having number 2 passivates itself. All other threads, except of the main one, passivate themselves again because they find that they are not intended to run. The main thread removes just executed event from the calendar and exits the `step()` function. One simulation step has just completed.

2.4 Life of a Process

Every Java thread has its own `run()` method that the user can (and should) overwrite in his subclasses to give his threads a specific behavior. Normally, this method does nothing and a thread which has just been started immediately terminates. Because J-Sim's class `JSimProcess` is a subclass of the `Thread` class, the `run()` method can be overwritten in a convenient way to fulfil the user's needs. However, this would be quite complicated for him because every J-Sim process must take care of the following things:

- At the very beginning of the `run()` method, every process must passivate itself. This is due to the fact that a part of a process' life can be executed only upon its simulation's request (activation by a signal from the `step()` method), not in a wilful manner as Java threads usually run.
- At the end of the `run()` method, the process must let the simulation (`myParent`) know that it is dying and cannot be run anymore.
- Between these two points, there must be a mechanism allowing the thread to finish in the usual way when it is suspended in its `hold()` or `passivate()` method and the simulation is shutting down – its `shutdown()` method is called.

- There must be a mechanism preventing the thread from collapsing when the user commits an illegal operation which would normally destroy the thread.

Since these requirements are always present and it is quite complicated to fulfil them from the user's point of view, the following strategy was applied:

- The `run()` method is marked as `final` which means that the user cannot overwrite it in his subclasses. This method invokes `getReady()` at the beginning which passivates the process. At the end, the `finish()` method is invoked to let the simulation know that this process is dying. In between, the `life()` method is called.

- In `JSimProcess`, the `life()` method is empty. Because this method is not marked as `final`, the user can (and should) overwrite it. This method represents the life of the user's process.

2.4.1 Two Kinds of Death

Taking a deep look at the possible strategies of processes' lifes, one will find that there are several of them:

- *Finite processes* which finish their `life()` method by reaching its end. These processes usually contain a few life parts divided by the `hold()` function. Their death can be called *natural* because they decide to die voluntarily.

- *Infinite processes* which contain a neverending loop inside their `life()` method. Such processes never finish voluntarily and the simulation must kill them when its `shutdown()` method is called. Because the `shutdown()` method is invoked outside of any simulation step, typically at the end of the main program, all such processes are suspended in the `hold()` or `passivate()` method and must be *killed*.

- *Finite processes* which have not had a chance to finish their lives because too few simulation steps has been executed before the `shutdown()` method is called. These processes are suspended in the `hold()` or `passivate()` method and must be *killed* too.

- *Finite or infinite processes* which have not been run yet at all. These processes are suspended in the `getReady()` method and must be *killed* too.

The second way of process termination – *killing* – requires a special mechanism which is discussed in the next chapter.

2.4.2 Killing a Process

As already known, the technique of process killing is used when a process is suspended in `hold()`, `passivate()` or `getReady()` and its simulation tries to destroy it. As mentioned in chapter 2.1.3, every suspension (a call to `aLock.wait()`) must be wrapped into a `try` block followed by an exception handler catching all instances of `InterruptedException`.

Since it is not recommended to use deprecated methods such as `stop()`, the only possibility how the simulation can interrupt a process is to invoke the `interrupt()` method: `aProcess.interrupt()`. Because the simulation knows about all processes belonging to it, it can interrupt all of them.

When a process is interrupted while being suspended, an instance of class `InterruptedException` is thrown out from the body of the `wait()` method and is caught in the exception handler which follows the mandatory `try` block. It is an easy job to set a flag saying that the process should die to `true`. The corresponding piece of code follows:

```
try
{
 setProcessState(STATE_PASSIVE);
 ourCommonLock.wait();
}
catch (InterruptedException e)
{
 shouldTerminate = true;
}
```

But another problem appears: The methods `hold()` and `passivate()` cannot be just exited. These methods are called from the `life()` method which always contains user's code. And he has no interest in taking care about exiting the `life()` method when it is necessary, i.e. in testing the flag of interruption. Therefore, a similar principle like in Java Core API is used: an exception is thrown out, which is not caught in the `life()` method but is caught in the `run()` method. Although this principle has been deprecated in Java Core API because it is possible to catch the exception and force the dying thread to stay alive, it suits its purpose very well. For the detailed reasons of deprecation, see [Lea], page 175. Although `getReady()` is not invoked from `life()`, it uses the same principle to unify the methods of process killing used. Here is the corresponding piece of code:

```
if (shouldTerminate)
 throw new JSimProcessDeath();
```

Since `JSimProcessDeath` is a subclass of `RuntimeException`, it need not be enumerated in the list of exceptions thrown out from a method. Therefore, the headers of methods that this concerns look as follows:

```
private final void getReady()
protected void life()
protected final void passivate()
     throws JSimSecurityException
protected final void hold(double deltaT)
     throws JSimSecurityException
```

Note that `JSimSecurityException` has nothing to do with the principle of process killing, it is just a way how to tell the user that he tries to commit an operation which is not allowed.

A little summary of `JSimProcessDeath` 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()`

2.4.3 Life of a Process Illustrated

The `run()` method, complexly described in previous chapters, is depicted in figure 5 on page 23.

The thick black arrow leading from `getReady()` to `life()` represents the moment when the process is selected for the first time by the simulation to run. The second thick black line leading from `life()` to `finish()` represents the process' natural death, i.e. the moment when the process reaches the end of `life()` by its own decision.

As it can be seen, there are three points where the process can be killed by its simulation: in `getReady()`, `hold()` and `passivate()`. It does not matter that the `hold()` and `passivate()` methods are not invoked directly from `run()` because the `JSimProcessDeath` exception is able to propagate through the `life()` method since the user does not care about it. The exception is caught in the handler, placed between the `life()` and `finish()` methods. Then, the execution continues by invoking `finish()`, as usual.

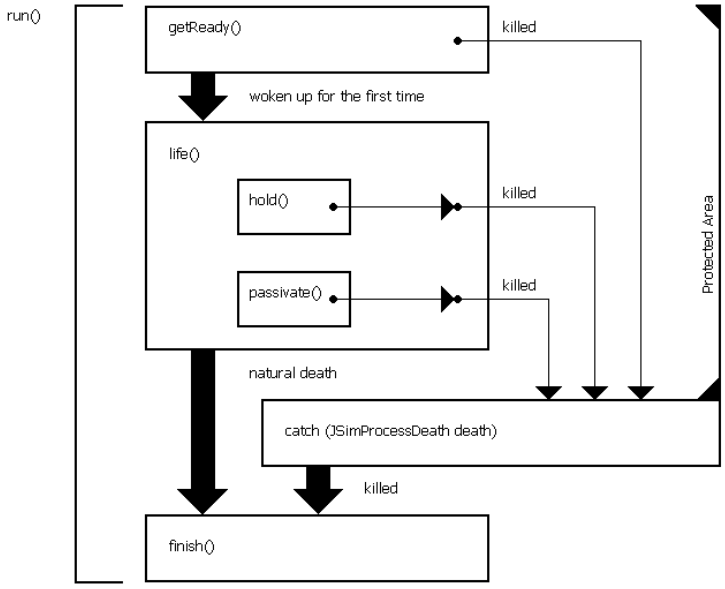

Figure 5: Life of a Process

2.4.4 Possible Danger of Catching `JSimProcessDeath`

The solution used by J-Sim to kill processes contains one hidden danger. Nothing can prevent the user from catching all `JSimProcessDeath` exceptions in the body of his `life()` method. He can do that if he uses at least one `hold()` or `passivate()` method in his `life()` because this exception is thrown out by them. In that case, the user is fully responsible for terminating the process and J-Sim has no other possibilities to do it. The following list contains possible solutions how to treat `JSimProcessDeath` exception in the `life()` method, beginning with the best one and ending with the worst one:

1. Ignore the existence of `JSimProcessDeath` completely. Neither write any exception handler nor add this exception to the list of exceptions thrown out by this method. This is the recommended solution.

2. Write an exception handler catching `JSimProcessDeath` but re-throw it from there. Again, this exception need not be in the list of exceptions thrown out by `life()`. An example follows:

```
  . . .
 } // try
 catch (JSimProcessDeath death)
 {
  // Some useful action(s) here...
  throw death; // This is important!
 } // catch
} // life()
```

3. Write an exception handler catching `JSimProcessDeath`, not re-throw it, but store the flag of need to terminate to a variable. At least once per cycle that the `life()` method usually contains check this variable and exit `life()` immediately. An example follows:

```
 while (!terminateLater)
 {
  try
  {
   // Process' life, including hold()s and passivates()s
  } // try
  catch (JSimProcessDeath death)
  {
   // Some useful action(s) here...
   terminateLater = true; // Setting the flag...
  } // catch
 } // while
} // life()
```

   However, if there are more than one `hold()` or `passivate()` methods called from inside the cycle, this solution will cause a *deadlock* because no process will wake up this suspended thread anymore.

4. Write an exception handler catching `JSimProcessDeath` and ignore that this exception was caught. This solution will inevitable lead to deadlock if the process invokes at least one `hold()` or `passivate()` after catching the exception. The reasons are the same as in previous point.

2.4.5 Prevention from Collapses

During execution of the process' life, number of runtime exceptions[9] can be thrown, for example `NumberFormatException` if the user tries to convert a badly formatted string to number.

These exceptions would normally propagate from `life()` through `run()` to JVM which would destroy the affected thread. Because this thread is the only running one at the moment, its destruction would *inevitably lead to deadlock*. All other threads, including the main thread, would stay suspended and the only visible result would be a "hang-up".

To avoid this situation, another exception handler must be inserted into the `run()` method, which will catch all exceptions caused by the user's programming mistakes. After catching the exception, the process continues as a normally terminated process (see natural death in chapter 2.4.1), i.e. by calling the `finish()` method.

2.5 A Deep View of `JSimSimulation`

In this chapter, a presentation of the `JSimSimulation` class in deeper detail with focus on some important fields and methods it provides for its processes and for the user will be given. Not all of them will be mentioned, only those of sufficient importance.

`JSimSimulation` is one of J-Sim's fundamental classes. It is inherited from `Object` and thus has no special pre-programmed behavior or services to offer. Only this class is able and allowed to execute the simulation in the step-by-step manner mentioned in chapter 1.1. An instance of this class must be present in every J-Sim simulation program. If it is not, it is not possible to create any process and queue (see chapters 2.6 and 3). It is not necessary to create a new subclass of `JSimSimulation`. It is a ready-to-use class whose instances are fully functional.

Here is a list of principal tasks that every simulation object is given:

- The simulation object has a list of processes which must be updated whenever a new process is created or a process dies.
- The simulation object has a *calendar of events* (instance of `JSimCalendar`) which must be updated whenever a process is activated (`activate()`), temporarily suspended (`hold()`), or cancelled (`cancel()`).

[9] Runtime exceptions are instances of `RuntimeException` or its subclasses and are not usually caught. They can raise at almost any point of a Java program.

- The simulation object must keep track of *simulation time*. The initial simulation time is 0 and it changes discretely with every step executed.
- The simulation object must know which of its processes is currently running, if a simulation step is being executed.
- The simulation object is responsible for switching between console mode and graphics mode of output – method `runGUI()`[10].
- To the user, the simulation object offers the possibility to execute one simulation step. The `step()` method provides this service.

2.5.1 Information about Processes

Every instance of `JSimSimulation` is equipped with a queue of `JSimProcessInfo` elements where it stores information about processes present in the simulation. The name of the queue is `infoQueue`. Whenever a new process is added to the simulation, a new `JSimProcessInfo` object is created and inserted to the queue. `JSimSimulation` class provides a method called `addProcess()`, taking one parameter – a reference to the process to be added. This method is called implicitly from the constructor of `JSimProcess`, therefore the user need not take care about that.

There are two methods deleting processes from the queue: `deleteProcess()` and `deleteAllProcesses()`. The difference between them is that `deleteProcess()` deletes just one process from the queue, whose number is specified as parameter of the method, while `deleteAllProcesses()` destroys the whole queue, sending all processes in the queue an interrupt signal. Why is the signal sent in `deleteAllProcesses()` but not in `deleteProcess()`? The answer is easy: `deleteProcess()` is called by the process itself from its `finish()` method when it is dying by a natural death while `deleteAllProcesses()` is called from simulation's `shutdown()` method when there are some threads suspended. It is thus necessary to kill them by interrupting their suspension. What follows after the interrupt signal is received is described in chapter 2.4.2.

2.5.2 The Calendar

The calendar (a variable called `calendar`) is an instance of the `JSimCalendar` class. It is nothing more than a queue of `JSimCalendarItem` elements. Every such element holds information about one event: time that the event is scheduled for – `time` – and number of the process which will be run at that

[10]GUI stands for Graphical User Interface.

time – `processNum`. The calendar provides some important methods that the simulation uses:

1. `addEntry()` adds a new element to the queue. Absolute time of the newly added event and a process' number must be specified.
2. `deleteEntries()` deletes either the first or all events from the calendar concerning a process. The process' number and the choice "*first/all*" must be specified.
3. `getFirstProcessNum()` returns the number of the process which is pointed to by the first element in the queue.
4. `getFirstProcessTime()` returns the time of the first event in the calendar. Process obtained in point 3 will be run at this time.
5. `jump()` deletes the first event from the calendar and sets the beginning of the queue to the following element. This method is invoked from the simuation's `step()` function when a step is completed.

Since processes do not know anything about the existence of the calendar they cannot use the methods mentioned above. Instead, `JSimSimulation` offers them methods called `addEntryToCalendar()` and `deleteEntriesInCalendar()` to insert and delete events. These functions are marked as `protected` and therefore can be invoked only from `JSimProcess`, not from its subclasses not being in the package `cz.zcu.fav.kiv.jsim`. They are used in `activate()`, `cancel()`, and `hold()` when processes need to create or delete an event.

2.5.3 The Simulation Time

The simulation time, stored in variable `time`, is a floating-point number which changes just before a simulation step starts to be executed. As it can be seen in figure 2 on page 4, the value of the change is not constant and cannot be predicted in any way unless the time of the next event in the calendar is known.

There is only one place in the code of `JSimSimulation` where the simulation time is changed: it is in the `step()` method when the new time is taken from the calendar and before the main thread switches to a selected process:

```
time = calendar.getFirstProcessTime();
```

Although it cannot be set by anybody else than the simulation, it can be read by everybody: processes, user – outside the `step()` function, etc. Processes can obtain the simulation time by calling `myParent.getCurrentSimulationTime()`, the user can obtain it by calling `x.getCurrentsimulationTime()` where `x` is a simulation object created in the main program.

2.5.4 Switching to Graphics Mode and Back

Every J-Sim simulation is able to run in two modes: in console mode and graphics mode. Implicitly, it is the console mode that the simulation runs in and the user must use a method called `runGUI()` to switch to graphics mode. Because it is not as trivial of an operation as it seems to be at the first sight, it must be explained here.

As stated in chapter 2.2, every simulation is able to create its own simulation window which is an instance of the `JSimGUIMainWindow` class. The internals will be discussed in chapter 2.7, while here the explaination of the relationship between the simulation and its window will be given.

To create a new window (an instance of `JSimGUIMainWindow`) and open it using its `show()` method, one will find that the window is visible and properly functionning but the `show()` method does not suspend the calling thread for the time the window is open. Instead, four[11] new threads are created to handle the user input (mouse, keyboard) and perform graphical operations. The calling thread continues running without waiting for the window's closure. This is the fundamental problem that must be dealt with.

The solution is obvious if figure 6 on page 29 is examined. It is exactly the same figure that was presented on page 14.

If the main thread after window's creation is suspended until the time the window is closed, the desired behavior will be obtained. To be able to do that, a new lock must be introduced, shared by the simulation object and the window. This lock is called `graphicLock` both in `JSimSimulation` and in `JSimGUIMainWindow`. Again, the lock is physically created only once (in the simulation's constructor) and the simulation window has only a reference to it. The mechanism of switching is less complicated than the mechanism of process switching but it will be explained anyway.

When the main thread creates the window and opens it, it must passivate itself until the main window sends it a wake-up signal. Again, the `wait()` method of the shared lock is used:

```
synchronized (graphicLock)
{
 mainWindow = new JSimGUIMainWindow(this);
 while (windowOpen)
 {
  try
  {
   graphicLock.wait();
```

[11] Using JDK 1.2.1 under Solaris 8, maybe the number can change.

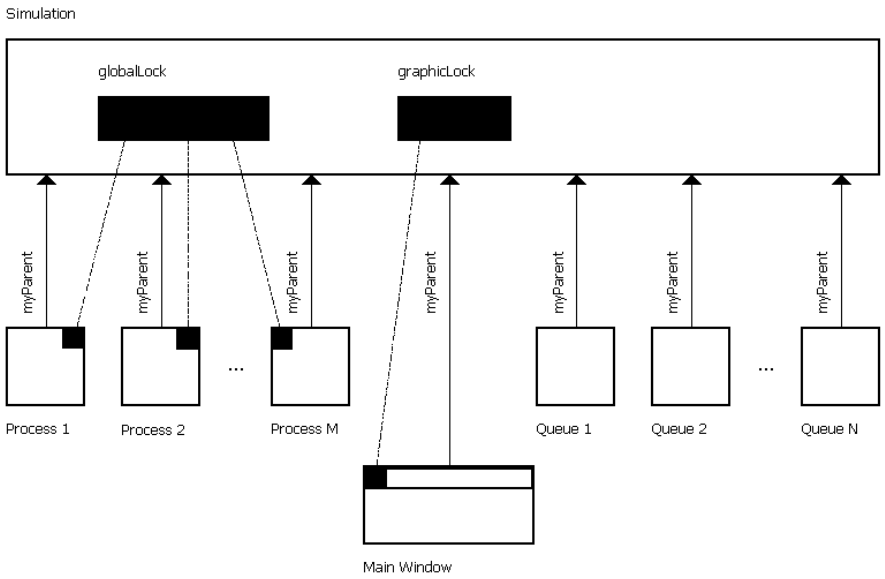

Figure 6: Synchronizing the Simulation with the Main Window

```
  }
  catch (InterruptedException e)
  {
   windowOpen = false; // We must get out
  }
 } // while
} // synchronized
```

When the main thread is woken up, it checks the value of `windowOpen`. If it is `false`, it exits the `while` cycle and consequently exits the `runGUI()` method. At this point, `windowOpen` should never be `true` again because the only sender of the wake-up signal is the main window which always sets this variable to `false` before the signal is sent. The following is the corresponding piece of code in `JSimGUIMainWindow`'s event handler:

```
synchronized (graphicLock)
{
 myParent.windowIsClosing();
 graphicLock.notifyAll();
 hide();
```

```
} // synchronized
```

Since the window itself cannot set the value `windowOpen` to `false` because it is a private variable, it must use the `windowIsClosing()` method of the simulation – `myParent`. Then, it sends the wake-up signal to all threads suspended at the lock. However, there is only one such a thread – the main one. After that, the window is hidden.

Once a graphics window using AWT[12] services is created, Java creates some auxiliary threads, as mentioned above. These threads are not destroyed when the window is closed, they cannot be destroyed by sending them the interrupt signal – `aThread.interrupt()` either, because this signal can be (and probably is) ignored by them. Since the `Thread.destroy()` method has not been implemented[13] there is no other way to prevent the program from hanging up unless the `System.exit()` method is used to exit back to operating system. Because it is done in the `shutdown()` method which is always called at the end of the program, neither the simulation nor the user are affected.

2.5.5 The `step()` Method in Detail

For the user, the `step()` method is the only really useful simulation's method. Although the principle of process switching, managed by this method, has been already discussed in chapter 2.3.2, a complex and detailed look at the method, explaining all possibilities that can occur during its execution, will be given.

First of all, the simulation must start all processes which were added to the simulation during last step. Therefore, `startNewProcesses()` is invoked at the very beginning of the method. After that, the execution differs according to the state in which the simulation is. The state is stored in a variable called `simulationState`. Every simulation can be in one of the three following states:

- *Not-started-yet* state. The simulation is in this state if no step has been executed yet. Its respective constant is `SIMULATION_NOT_STARTED`.
- *In-progress* state. The simulation is in this state if at least one step has been executed and the simulation has not terminated yet. Its respective constant is `SIMULATION_IN_PROGRESS`.

[12] Abstract Window Toolkit, a library for GUI programs provided by Java

[13] And will probably never be, according to [Java].

- *Terminated* state. The simulation gets to this state if there are either no processes or no calendar events and the user tries to do a step. Its respective constant is `SIMULATION_TERMINATED`.

If the simulation has not been started yet, the `step()` method checks whether there is at least one process within the simulation. If there is not, the simulation sets the state to terminated and does not continue anymore, otherwise it sets the state to running (in progress) and continues normally.

If the simulation has already terminated, the `step()` method does not do anything and simply exits.

In the usual case (`simulationState` is equal to `SIMULATION_IN_PROGRESS`), the `step()` method takes from the calendar the first process' number and time and store these values to `runningProcessNum` and `time`. Then it checks whether the selected process really exists (it must be found in the queue of processes inserted into the simulation – `infoQueue`) and has not finished yet – `isAlive()` must return `true`. When one of the tests fails, it is necessary to repeat the selection. If there are no more events in the calendar or `infoQueue` is `null` (there are no processes), the simulation is terminated.

Finally, when a process is selected, the switching mechanism, described in chapter 2.3.2, is used, the selected process is woken up and running and the main thread is suspended. When the process switches back to the main thread, the event just executed is removed from the calendar by calling `calendar.jump()`.

The method returns a `boolean` value saying whether it is possible to execute another step or not.

2.6 A Deep View of `JSimProcess`

Like `JSimSimulation`, `JSimProcess` is one of J-Sim's fundamental classes. It is a subclass of the `Thread` class and therefore inherits all its possibilities, mainly the possibility to be executed concurrently to other threads. New subclasses of `JSimProcess` should be always created since the `life()` method, representing the life of the process, is initially empty in `JSimProcess`. The only necessary changes in such a subclass are:

- new constructor, since `JSimProcess`' constructor cannot be invoked automatically because it has two parameters;
- new implementation of the `life()` method.

2.6.1 States of a Process

During its life, every process can be in one of four different states:

- *new* – A process is in this state typically for a very short time between it is created in one simulation step (or before any simulation step) and it is started in the following simulation step, when the simulation invokes `startNewProcesses()`. Once a process is started and enters its `getReady()` method where it is suspended, the process becomes passive and cannot become new anymore. The corresponding constant is `STATE_NEW`.

- *passive* – A process is in this state for most of its life. Typically, it switches to this state whenever `hold()` or `passivate()` are invoked and it stays in this state until the process is woken up by the simulation, exits one of these methods and starts to execute a part of its life. When it invokes `passivate()` or `hold()` again, it switches back to the passive state. The corresponding constant is `STATE_PASSIVE`.

- *active* – A process switches to this state when it is woken up by the simulation and starts to execute a part of its life. It can switch back to the passive state by invoking `hold()` or `passivate()` and to the terminated state by reaching the end of its `life()` method. Only one of all processes present in the simulation can be in the active state while all others are new, passive or terminated. The corresponding constant is `STATE_ACTIVE`.

- *terminated* – A process switches to this state automatically when it reaches the end of its `life()` method or is killed by its simulation. Once it gets to this state, it cannot switch to any other state or be restarted. The corresponding constant is `STATE_TERMINATED`.

Unlike in C-Sim, there is not any state called scheduled. The reasons for this will be discussed in chapter 2.6.2.

2.6.2 The Concept of Over-scheduling

In C-Sim, every process can have at most one event in the calendar. In J-Sim, a different approach was chosen. A process can have unlimited number of events in the calendar. This implies the possibility to schedule an already scheduled process. All already present events belonging to the process are kept in the calendar and a new one is inserted there. There are three possibilities how a new event can be added to the calendar:

- Before the simulation is started or between successive simulation steps, a process can be scheduled if its `activate()` method is invoked:

```
aProcess.activate(5.67);
```

However, the time specified as parameter must be at least equal to or greater than the current simulation time.

- A process can be scheduled by another process in its `life()` method. The scheduling process must have a reference to the process being scheduled in order to be able to invoke its `activate()` method.

- A process can schedule itself during its life by invoking its `hold()` method. This will suspend it for a certain time (the only parameter of the method) and the process will be woken up by the simulation after that period. The process does not need any reference because it calls its own method, inherited from `JSimProcess`.

```
hold(0.123);
```

Because these methods can be combined, there can be more than one event in the calendar, belonging to a process. It is guaranteed that the process will be woken up at all desired times. However, it is not guaranteed that the process will be passivated for the exact time given as parameter to the `hold()` method. Another process can activate the process (before or after the process calls `hold()`) at a time lying in the "passive" interval and the process must be woken up at that time and therefore, must exit the `hold()` method. In fact, this allows us to "clone" process' life because this method doubles the number of process' events in the calendar.

Therefore, over-scheduling can be considered as extremely useful when the user needs to add process scheduling times from outside but also very dangerous if the user depends on the length of time gaps between two successive steps of process' life.

Since there can be any non-negative number of events belonging to a process, there is no need to register whether the process is scheduled or not. In fact, a process can be scheduled (or not scheduled) in any of the four previous states mentioned in chapter 2.6.1:

- In the state called new, a process can be scheduled if another process created it and "activated" (`aProcess.activate(time)`) in the same simulation step and the step has not finished yet or this happened outside any simulation step (typically before the simulation starts).

- In the state called passive, a process can be scheduled if another process has activated it (`aProcess.activate(time)`) and its activation time has not come yet.

- In the state called active, a process can be scheduled if there are at least two of its events it the calendar: the currently running one and another one. This is the situation of over-scheduling. One of the events could be inserted by the process itself (`hold(time)`), and the rest of them by other processes (`aProcess.activate(time)`).

- Even when a process is terminated, it can be scheduled by other processes. After a process terminates, its code cannot be executed, however the object of the process still exists. When the simulation tries to perform an action on a terminated process, it finds that the process cannot run anymore and ignores the corresponding event.

2.6.3 Five Principal Methods

There are five principal methods provided by the `JSimProcess` class for the user: `life()`, `activate()`, `cancel()`, `hold()` and `passivate()`. The first one is intended to be overwritten in user's subclasses, the four others are marked as `final` and therefore are unchangeable.

The `life()` method represents life of a process. The user can put any possible code in it, also code creating new object, including new processes. Initially, this method is empty and it has no sense not to overwrite it in user's subclasses. This method is called automatically from `run()` and should never be called explicitely by user. To avoid this, the method is marked as `protected` and thus can be called only from methods of the same class (`JSimProcess`) or of the same package (`cz.zcu.fav.kiv.jsim`). The life of a process can be divided into several parts, using methods `hold()` and `passivate()`. The method's header is:

```
protected void life()
```

The `activate()` method inserts a new event into simulation's calendar. The 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 time of activation. The method is marked as `public` and thus can be called by any process or even the main thread. The method's header is:

```
public final void activate(double when)
```

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 – the method is marked as `public`. The method's header is:

```
public final void cancel()
```

The `hold()` method temporarily suspends the calling process and wakes it up after the time specified as method's parameter. In fact, it adds a new event to the calendar and then passivates the process. A process can invoke this method on itself only, not on another process. If it tries to do so, a `JSimSecurityException` is thrown out. The method's header is:

```
protected final void hold(double deltaT)
     throws JSimSecurityException
```

The `passivate()` method suspends the calling process without creating a new event for it. If the process is not over-scheduled now or woken up later by another thread, it will stay passive forever. A process can invoke this method on itself only, not on another process. If it tries to do so, a `JSimSecurityException` is thrown out. The method's header is:

```
protected final void passivate()
     throws JSimSecurityException
```

2.7 A Deep View of `JSimGUIMainWindow`

The `JSimGUIMainWindow` class provides a user-friendly method how to run a simulation. Instead of executing steps in the way prescribed in the main program, the user can interactively decide how to proceed with the simulation. The window provides the following possibilities:

- to execute one simulation step;
- to execute a given number of steps;
- to execute an unspecified number of steps, limited by a time value. It will execute as many steps as necessary to reach (or overpass) the time limit;
- to close the window and continue to execute the simulation in console mode.

In order to handle the user's input (mouse and keyboard input) properly, it creates and registers two event listeners – instances of `JSimGUIMainWindowWindowAdapter` (for events generated by the window's frame, such as the window's closure) and `JSimGUIMainWindowActionAdapter` (for events generated inside the window, such as button presses). To describe the model

of event creation, propagation and handling, used in AWT[14] is beyond possibilities of this paper. A lot of information on this subject can be found for example at [Java]. The event listeners are the window's inner classes[15] and thus have unlimited access to all of the window's fields and methods. In fact, they do not do anything else except transformation of events, sent by AWT, to method invoking. Here is a list of the events and their corresponding methods, with a brief description:

- When the button "*Run one step*" is pressed, `actionRunOneStep()` is called. This method simply executes one simulation step and then exits. The execution is possible due to the fact that the window has a reference to the simulation it belongs to (`myParent`) and therefore calls `myParent.step()`. Look back at figure 3 at page 14.

- When the button "*Run N steps*" is pressed, `actionRunNumberOfSteps()` is called. This method reads a number inserted at input line `fieldNoOfSteps` and executes the same number of steps. If the simulation is terminated before the required number of steps is reached, the method does not proceed.

- When the button "*Run until time limit*" is pressed, `actionRunUntil-TimeLimit()` is called. This methods reads the time limit inserted at input line `fieldUntil` and executes as many steps as necessary until the simulation time is greater than or equal to the limit. If the simulation is terminated before the time limit is reached, the method does not proceed. The window can obtain current time by invoking `myParent.getCurrentTime()`.

- When the button "*Quit*" is presed or the window is about to be closed, `actionQuit()` is called. This method does not destroy the window, it only hides it and sends a wake-up signal to the main thread which is being suspended in the `runGUI()` method. Before doing so, it must inform the simulation that the window is being closed: `myParent.windowIsClosing()` is called. The principle of switching from the console mode to the graphics mode was fully described in chapter 2.5.4. As recalled from the previous chapter, the simulation and the window share a lock called `graphicLock` in order to be able to do the switching.

The main window has a text area covering most its surface. When the simulation runs in graphics mode, the output generated by `JSimProcess.message()`

[14]Abstract Window Toolkit

[15]More information can be found in [Her], chapter 15.

and `JSimProcess.error()` methods is redirected to this area. In the console mode, these methods print to `System.out` and `System.err`, respectively.

3 Queue Facilities

Although the Java language provides a broad spectrum of classes intended to be directly used as queues, J-Sim introduces two new ones, based on the philosophy of the Simula language and used by C-Sim, too. The new classes are named `JSimHead` and `JSimLink`.

In Simula, three classes exist: `LINKAGE`, `HEAD` and `LINK`. `HEAD` represents the head of a queue where various objects can be inserted. `LINK` represents an object holding user's information, supposed to be inserted into a queue. Both these classes are derived from `LINKAGE` which is a class having the ability to be linked, i.e. to be inserted into a queue. Since the `HEAD` class is derived from `LINKAGE`, it has this ability too; therefore, the user can construct queues having other queues as their elements. The user must derive a new class from `LINK` and put his information into it because instances of `LINK` are not supposed to hold any information. Therefore, the `LINK` class can never be used in a real application, but thanks to inheritance and polymorphism all parameters of operations can be typed as `LINK`.

In J-Sim, this principle was considerably simplified but it still has all Simula's possibilities. Although the `LINKAGE` class was not implemented at all, J-Sim's queues have ability to contain objects of any Java class. This is due to the fact that `JSimLink` does not keep user's information inside but has just a reference to the information. This reference is typed as `Object` which is superclass of all possible classes. Since `JSimHead` is a subclass of `Object`, as all classes, the user can create queues containing other queues. There is therefore no need to create subclasses of `JSimLink`, although it is possible (but much more complicated for the user).

A simple model of a queue (instance of `JSimHead`) containing four elements (instances of `JSimLink`) is depicted in figure 7 on page 39. Note that the elements of the queue do not necessarily hold information of the same type, i.e. instances of the same class.

Inside the queue, elements are organized in a double-ended bi-directional list and the queue holds information about both ends of the list (`head` and `tail`). Every element holds information about the queue it is inserted in, about its data and about the next and previous element in the list. An element can be inserted into one queue only; however, later it can be removed from the queue and inserted into another one.

Since the information held by an element is always an instance of a class, it is not possible to insert there variables of primitive data types like `int`, `float`, `double`, etc. Fortunately, it does not cause any problems since the Java

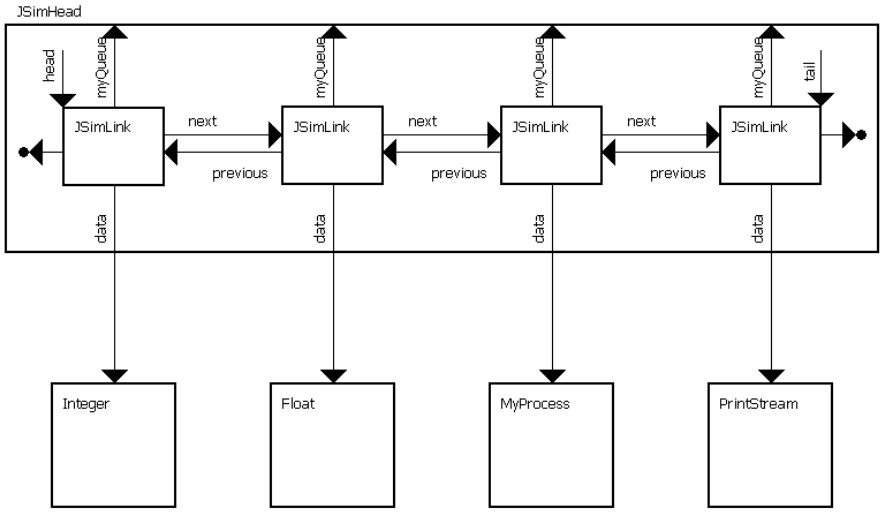

Figure 7: A Simple Queue Having Four Elements

language provides a class[16] for each of these primitive data types, being able to hold information about one variable of its respective type. The `Integer` class corresponds to the `int` type, `Float` to `float`, `Double` to `double`, etc.

The following subchapters will discuss internals of the two classes as well as the way how they cooperate.

3.1 Queues – The `JSimHead` Class

The `JSimHead` class can be considered as a container where data of various types are stored in a defined way. It provides some useful functions inherited from Simula's queue concept as well as some new ones added in J-Sim. Maybe it is surprising that it does not offer any possibility how to insert an element into the queue or how to remove it from the queue. In J-Sim, these services are provided by the elements themselves, not by the queue. They will be discussed in chapter 3.2.1.

[16]These classes are called "wrapper classes".

3.1.1 Simula-like Functions

Only five basic methods, taken from Simula, are offered by the `JSimHead` class:

The `empty()` method returns a logical value saying whether the queue is empty or not. The queue is empty if it contains no elements. The method's header:

```
public boolean empty()
```

The `cardinal()` method returns the number of elements currently present in the queue. The method's header:

```
public long cardinal()
```

The `first()` method returns the element which is at the beginning of the queue. If the queue is empty, `null` is returned. Note that it returns a reference to `JSimLink`; therefore, it is necessary to use other functions, described in chapter 3.2.2, to get the real data. The method's header:

```
public JSimLink first()
```

The `last()` method returns the element which is at the end of the queue. If the queue is empty, `null` is returned. Like in the previous case, it returns a reference to `JSimLink`; therefore, it is necessary to use other functions, described in chapter 3.2.2, to get the real data. The method's header:

```
public JSimLink first()
```

The `clear()` method removes all elements from the queue and sets queue statistics to their initial values. While removing the elements from the queue, they may be disposed. Whether they will be disposed or not depends on the number of links pointing to them. If there is at least one such link (except of this queue's links), the element will be preserved. Otherwise, it will be destroyed automatically by JVM. The method's header:

```
public void clear()
```

3.1.2 Other Useful Functions

Since J-Sim adds new ideas to Simula's queue concept, it provides some new methods making queues easier to use and more comprehensible. The new methods are as follows:

The `getFirstData()` method returns a reference to the data that the first element in the queue points to. If the queue is empty, `null` is returned.

Note that the user must always retype the result to the required class in order to have access to all fields and methods of the class. The method's header:

```
public Object getFirstData()
```

The `getFirstDataType()` method returns the name of the class that the first element's data object is instance of. The name returned is fully qualified, e.g. "`java.lang.Integer`". If the queue is empty, `null` is returned. The method's header:

```
public String getFirstDataType()
```

The `getLastData()` method returns a reference to the data that the last element in the queue points to. If the queue is empty, `null` is returned. Like in the previous case, the user must always retype the result to the required class in order to have access to all fields and methods of the class. The method's header:

```
public Object getLastData()
```

The `getLastDataType()` method returns the name of the class that the last element's data object is instance of. The name returned is fully qualified, e.g. "`java.lang.Integer`". If the queue is empty, `null` is returned. The method's header:

```
public String getLastDataType()
```

The `getCurrentTime()` method returns the current time of the simulation that the queue belongs to. This method is especially useful for statistics purposes. The method's header:

```
public double getCurrentTime()
```

3.1.3 Statistics Functions

Since the queueing theory introduces some queues' quantities that can be measured or theoretically computed, J-Sim provides functions for their computation, too. They are as follows:

The `getLw()` method returns the average length of the queue. The length is computed over a time interval whose left limit is the time of queue's creation (or the time when last `clear()` was invoked) and the right limit is the current time. This quantity's sign is L_W. The method's header:

```
public double getLw()
```

The `getTw()` method returns the average waiting time that the elements ever inserted into the queue have spent there. When `clear()` is called, this number is set to zero. This quantity's sign is T_W. The method's header:

```
public double getTw()
```

3.1.4 Managing Data

When an element wants to insert itself into a queue or to remove itself from a queue, it must cooperate with the queue the operation concerns. When such an operation is performed, the queue's statistics must be updated and sometimes, the head or the tail of the queue must be adjusted. The following methods are called exclusively from `JSimLink` and therefore can be marked as `protected`:

The `setHead()` method simply replaces the value of head by the value passed as parameter to the method. No other action is performed. The method's header:

```
protected final void setHead(JSimLink newHead)
```

The `setTail()` method simply replaces the value of the tail by the value passed as parameter to the method. No other action is performed. The method's header:

```
protected final void setTail(JSimLink newTail)
```

The `putAtHead()` method puts a new element at the head of the queue, doing some necessary actions in order to keep the queue in consistent state. The method's header:

```
protected final void putAtHead(JSimLink newHead)
```

The `putAtTail()` method puts a new element at the tail of the queue, doing some necessary actions in order to keep the queue in consistent state. The method's header:

```
protected final void putAtTail(JSimLink newTail)
```

The `incNoOfItems()` method increments an internal counter of elements present in the queue and updates `sumLw` which serves as intermediate result for computation of L_W. The method's header:

```
protected final void incNoOfItems()
```

The `decNoOfItems()` method decrements an internal counter of elements present in the queue and updates `sumLw` and `sumTw` which serve as intermediate results for computation of L_W and T_W. The method must know when the element being removed entered the queue in order to update `sumTw` correctly. This time is passed to the method as a parameter. The method's header:

```
protected final void decNoOfItems(double whenEntered)
```

3.2 Elements – The `JSimLink` Class

Instances of `JSimLink` are elementary units of information having the ability to be inserted into a queue (instance of `JSimHead`). Since they can hold any type of information, they are supposed to be used directly, thus there is no need to create their subclasses. This is a fundamental difference from C-Sim and Simula, where this was the only way how to insert data into a queue. Thanks to polymorphism, this possibility is still kept in J-Sim and will be discussed in chapter 3.2.3.

3.2.1 Simula-like Functions

The following methods, taken from Simula, take care about inserting an element into a queue or removing it from the queue in which the element is already inserted. All methods are supposed to be called in this way: `element.method(x)`.

The `into()` method inserts the element into a queue specified as parameter. The element is always inserted at the end of the queue. If the element is already inserted in a queue, an exception is thrown out. The method's header:

```
public final void into(JSimHead queue)
     throws JSimSecurityException
```

The `follow()` method inserts the element into the same queue that the parameter of the method is inserted in, and right after the parameter. Therefore, the parameter of the method will become element's "left neighbour". If the element is already inserted in a queue, an exception is thrown out. The method's header:

```
public final void follow(JSimLink otherItem)
     throws JSimSecurityException
```

The `precede()` method inserts the element into the same queue that the parameter of the method is inserted in, and right before the parameter.

Therefore, the parameter of the method will become element's "right neighbour". If the element is already inserted in a queue, an exception is thrown out. The method's header:

```
public final void precede(JSimLink otherItem)
     throws JSimSecurityException
```

The `out()` method removes the element from the queue it has been inserted in. If the element is not inserted in any queue, nothing happens. The method's header:

```
public final void out()
```

3.2.2 Other Useful Functions

Other functions, available in the `JSimLink` class, serve mainly to provide access to the data that the element only points to but does not contain directly.

The `getEnterTime()` method returns the time when the element was inserted into a queue. If it has not been inserted into any queue yet, it returns zero. The method's header:

```
public final double getEnterTime()
```

The `getDataType()` method returns the fully qualified name of the class that the data object pointed to by this element is instance of. The fully qualified name contains the name of the package the class belongs to, e.g. "`java.lang.Number`". The method's header:

```
public String getDataType()
```

The `getData()` method returns a reference to the data object. The reference is typed as `Object` so the user will always need to retype it to his desired type. The actual type can be obtained by calling the `getDataType()` method. The method's header:

```
public Object getData()
```

3.2.3 Direct Use versus Inheritance

Although J-Sim provides a very intuitive way how to insert data of any type (instances of any class) into a queue without inheriting new classes from `JSimLink`, some users may prefer the way they used in Simula or C-Sim.

Some advantages of the imlicitly used way should be mentioned:

- No new classes have to be created.
- Any data can be inserted in a queue, even data of different types in the same queue.
- The data type of every queue element can be determined dynamically and according to that, the data obtained from the queue can be retyped dynamically in the right way.

If the user, despite these advantages, tries to inherit his own subclasses of `JSimLink` and put his data directly to them, he should be aware of the following necessary steps that must be performed:

1. A new constructor must be created since `JSimLink` does not have an implicit one.
2. The `getDataType()` method must be overwritten since the old one will probably return `null` because the data object will be initialized to `null` or not initialized at all.
3. The `getData()` method must be overwritten because of the same reasons. It should return a reference to an object fully owned by the new subclass.
4. No other methods can be overwritten since they are all marked as `final`. However, new ones can be added.

4 System of Exceptions

In Java, exceptions are an undetachable part of the language. Their usage is mandatory which prevents programmers from ignoring them. Because it is a very useful mechanism to signal an error or an unpredicted situation to an upper level of execution, J-Sim introduces some new exception classes, too. They are as follows:

- `JSimException`
- `JSimInvalidParametersException`
- `JSimSimulationAlreadyTerminatedException`
- `JSimTooManyProcessesException`
- `JSimKernelPanicException`
- `JSimProcessDeath`

4.1 Standard Exceptions

4.1.1 `JSimException`

The `JSimException` class is an extremely important exception, although it does not have any instances. It is a basic class that all other exceptions (except for `JSimKernelPanicException` and `JSimProcessDeath`) are derived from. All its subclasses inherit a variable called `specific` where the exact reason why the exception was thrown out is usually stored. This class, as well as all its subclasses, provides a constructor whose one parameter is reserved for that purpose.

The `printComment()` method, which should be rewritten in every subclass, is supposed to inform the user in a comprehensible way about the general reasons why this type of exception is usually thrown out. It prints the message into a `PrintStream` which must be specified. The method's header:

```
public void printComment(PrintStream ps)
```

The `getSpecific()` method returns a string containing information about specific circumstances of this exception. This information should always be given to one of the two constructors available in `JSimException`. The method's header:

```
public String getSpecificInfo()
```

The two constructors allow either to specify a short description of the exception or to use a default one, but the specific reason must be always given to them:

```
public JSimException(String description, String param)
public JSimException(String param)
```

This class of exception and all its subclasses must be always caught or explicitly stated in the list of exceptions thrown out by a method.

4.1.2 JSimInvalidParametersException

The `JSimInvalidParametersException` class is inherited from `JSimException` and therefore keeps all its possibilities. It is thrown out whenever a method gets arguments that cannot be used because their values are invalid (e.g. a negative number) or are not given at all (e.g. a `null` reference).

In `JSimInvalidParametersException`, only one constructor is present, using an implicite description:

```
public JSimInvalidParametersException(String pars)
```

The `printComment()` method is overwritten in an appropriate way.

4.1.3 JSimSimulationAlreadyTerminatedException

An instance of this exception is thrown out from `JSimProcess`' constructor when the simulation that the new process is to be inserted in is already terminated. A simulation is terminated if there are no more processes or no more scheduled events.

In `JSimSimulationAlreadyTerminatedException`, only one constructor is present, using an implicite description:

```
public JSimSimulationAlreadyTerminatedException(String pars)
```

The `printComment()` method is overwritten in an appropriate way.

4.1.4 JSimTooManyProcessesException

An instance of this exception is thrown out from `JSimProcess`' constructor when the simulation cannot assign a new number to the newly created process because the internal counter of created processes has reached its upper limit. The upper limit is equal to `Long.MAX_VALUE` which is 9 223 372 036 854 775 807. Once a process is created, its number gets occupied forever and even if the process terminates, the number is not available anymore. Therefore, this

exception can be thrown out even if very few processes (e.g. 3) are currently running in the simulation, but the simulation has already contained nearly the limit of processes which have died.

In `JSimTooManyProcessesException`, only one constructor is present, using an implicite description:

```
public JSimTooManyProcessesException(String pars)
```

The `printComment()` method is overwritten in an appropriate way.

4.2 Nonstandard Exceptions

4.2.1 `JSimKernelPanicException`

Unlike exceptions discussed in chapter 4.1, the `JSimKernelPanicException` is a subclass of `RuntimeException` and therefore need not be caught or explicitly stated in the list of exceptions thrown out from a method. It is used when the J-Sim kernel (typically `JSimSimulation` or `JSimProcess`) finds an inconsistency in data which cannot be repaired. Throwing out this exception usually leads to deadlock because the currently running thread is the only active one in the system and the exception is propagated until the JVM which will finally kill it. However, the purpose of this exception is not to stabilize the system, the purpose is to let the developer know that there is a weak point in the system. The constructor has no parameters:

```
public JSimKernelPanicException()
```

4.2.2 `JSimProcessDeath`

The `JSimProcessDeath` exception is an internal J-Sim exception which should not be used by the user – it should neither be thrown nor caught. It is thrown out by a process itself when it gets a killing signal from its simulation. The exception can be thrown out from these methods: `getReady()`, `hold()` and `passivate()`. Because `hold()` and `passivate()` are invoked from `life()` and this method does not catch the exception, it is propagated until `run()` where it is finally caught. Detailed information about `JSimProcessDeath` can be found in chapter 2.4.

`JSimProcessDeath` is a subclass of `RuntimeException` and has one constructor without parameters:

```
public JSimProcessDeath()
```

`JSimProcessDeath` is an equivalent of Java's `ThreadDeath` used when a thread is killed using its `stop()`[17] method.

[17]This method has been deprecated.

5 Other Important Features

In this relatively short chapter, some auxiliary functions, provided by J-Sim in addition to its standard services, will be shortly described. These services are offered by the `JSimSystem` class as its static methods, thus there is no need to make any instances of this class. All the methods mentioned below can be invoked like this: `JSimSystem.method(parameters)`.

5.1 Generators of Random Numbers

In J-Sim, following random-numbers generators are available:

- generator of random numbers having exponential distribution;
- generator of random numbers having uniform distribution;
- generator of random numbers having normal (Gaussian) distribution;
- generator of random boolean values.

The exponential-distribution random numbers are generated by a method called `negExp()`. It takes one argument `lambda` which is the theoretical parameter λ of the generator. The method's header:

```
public static double negExp(double lambda)
```

The uniform-distribution random numbers are generated by a method called `uniform()`. It takes two arguments `a` and `b`, representing boundaries of the interval $< a, b >$, from which the random numbers are taken. The method's header:

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

The normal-distribution random numbers are generated by method `gauss()` which takes two arguments `mu` and `sigma`. These arguments represent the theoretical mean value μ and the theoretical variance σ. The method's header:

```
public static double gauss(double mu, double sigma)
```

The `draw()` method returns boolean values accordingly to the probability p given as parameter; `true` is returned with probability p, `false` with probability $1 - p$. The value of the argument should be within the interval $< 0, 1 >$. If it is less than 0, `false` if always returned. If it is greater than 1, `true` is always returned. The method's header:

```
public static boolean draw(double probability)
```

5.2 Version Functions

Every J-Sim version has its major, minor and patch version number. They can be obtained using the following methods:

- `public static int getVersionMajor()`
- `public static int getVersionMinor()`
- `public static int getVersionPatch()`

To obtain complete J-Sim version, including the information about language and country, the `getVersion()` method can be used. It returns a string containing all available information, e.g. "`J-Sim version 0.1.1 en_US`".

6 Programmer's Cookbook

Despite all theoretical information presented in previous chapters, the user may still be confused and not able to use J-Sim at all. This chapter should help him by giving him practical information on how to build up a simulation program efficiently in the shortest possible time.

6.1 Creating a Simulation

The simulation is usually created in the `main()` method of a class founded especially for this purpose. Please note that there is no need to inherit a new subclass of `JSimSimulation`, this class can be used directly. Start with a file called `Example.java`. In its `main()` method, we will create a simulation named "*First simulation*". Because some processes are to be added and activated in the future, all exceptions that could possibly be thrown out must be caught. The best way how to do it is to write just one exception handler at the end of the `main()` method, catching all instances of `JSimException` and its subclasses. In its corresponding `finally` block, the user must shut down the simulation. If the user forgets to do it, the program will hang up upon termination since there will be some suspended threads. Note that the class must be named `Example` to have the name identical with the name of the file (without the extension, of course). Also, all necessary classes from the J-Sim package must be imported.

```
import cz.zcu.fav.kiv.jsim.*;
public class Example
{
 public static void main(String[] args)
 {
  JSimSimulation        simulation = null;
  try
  {
   simulation = new JSimSimulation("First simulation");
   // Some code will be inserted here:
   // ......................
  } // try
  catch (JSimException e)
  {
   e.printStackTrace();
   e.printComment(System.err);
  } // catch
```

```
  finally
  {
   simulation.shutdown();
  }
 } // main
} // Example
```

Trying to compile this file now (`javac Example.java`) will result in an error saying that `JSimException` is never thrown in the body of the `try` block. This is just a preparation of the source code for further improvement, please wait with the compilation until some processes are added into the simulation.

6.2 Creating a Process

Several necessary steps must be done in order to create a process:

1. A new class must be founded, extending the class `JSimProcess`. This class can be in its own separate file (and then it must be marked as `public`) or in the file where the main program is placed (`Example.java`). Then it must not be marked as `public`. Example (file `MyProcess.java`):

   ```
   public class MyProcess extends JSimProcess
   { // .....
   } // class MyProcess
   ```

2. The new class must have a constructor, invoking `JSimProcess`' constructor as its first command. This is due to the fact that `JSimProcess` has no constructor without parameters and therefore, this constructor cannot be invoked implicitly. Because `JSimProcess`' constructor requires two parameters (a name and a reference to a simulation), it is good to have these two values as parameters of the new constructor, too. `JSimProcess`' constructor throws out three exceptions:

   - `JSimSimulationAlreadyTerminatedException`;
   - `JSimInvalidParametersException`;
   - `JSimTooManyProcessesException`.

   It makes no sense to catch them inside the new constructor; therefore, they should be listed in the throws clause. Some new parameters can be added to constructor's header, as well as new code to its body. Example (still file `MyProcess.java`):

```
public MyProcess(String name, JSimSimulation sim,
                 ... new parameters ...)
     throws JSimSimulationAlreadyTerminatedException,
            JSimInvalidParametersException,
            JSimTooManyProcessesException
{
 super(name, sim);
 // ..... new code
} // constructor
```

3. The `life()` method of the new process must be overwritten. There are no special constraints or recommendations about that. Example (still file `MyProcess.java`):

```
protected void life()
{
 message("Hello!");
} // life
```

4. A new instance of `MyProcess` must be created in the main program after the point where the simulation is created. It will be automatically added to the simulation. Example (file `Example.java`):

```
public static void main(String[] args)
{
 JSimSimulation  simulation = null;
 MyProcess       process = null;

 try
 {
  simulation = new JSimSimulation("First simulation");
  process = new MyProcess("My process No 1",
                          simulation,
                          ... other parameters ...);
 } // try
 catch (JSimException e)
 {
  e.printStackTrace();
  e.printComment(System.err);
 } // catch
 finally
```

```
 {
  simulation.shutdown();
 }
} // main
```

Now, compile both files and run the example:

1. `javac MyProcess.java`
2. `javac Example.java`
3. `java Example`[18]

Now, it is possible to compile both files without errors, however, when the example is run, nothing happens. This is due to the fact that the process was not activated and the simulation was not asked to run.

6.3 Running the Simulation

When a process is created, it should be activated. A process can be activated by the main program or by another process. However, if there is just one process within the simulation, or no process has been activated yet, the latter case is not possible. A process will be activated at a certain time when its `activate()` method is called with the required time as parameter. Example (file `Example.java`):

```
process = new MyProcess("My process No 1",
                        simulation,
                        ... other parameters ...);
process.activate(1.2345);
```

Once a process is activated, it is able to care about its own scheduling using the `hold()` method; therefore, no other activation is needed.

Now, attention should be turned to the simulation and its execution. Every simulation can be principally executed in two ways: a pre-programmed way and an interactive way when the simulation's execution is driven by the user. The second one will be explained as first, thanks to its enormous ease. After the point where processes are activated, a call to the simulation's

[18] Without any extension!

`runGUI()` method must be inserted. The method will switch the simulation to the graphics mode and will wait until the user presses the "*Quit*" button. Example (file `Example.java`):

```
simulation = new JSimSimulation("First simulation");
process = new MyProcess("My process No 1",
                        simulation,
                        ... other parameters ...);
process.activate(1.2345);
simulation.runGUI();
```

The second possibility is based on calls to the `step()` method. Usually, it is done in a `while` or `for` cycle as shown here:

```
while (simulation.step() == true)
 ;
```

The `step()` method returns `true` if the simulation is not terminated, i.e. it can do another step. This is a potentially dangerous construction since the simulation can contain an infinite process. Such a process will make the simulation infinite too; therefore, the `step()` method will never return `false` and the program will never finish. There are two ways how to avoid this. Limit the execution either by time or by a number of steps to be executed.

The time limitation can be realized in the way shown below, using the simulation's method `getCurrentTime()`:

```
while ((simulation.getCurrentTime() < 100.0)
      && (simulation.step() == true))
 ;
```

The second way of limitation is usually realized by a `for` cycle which has a counter of steps executed:

```
for (int i = 0; (i < 25) && (simulation.step() == true); i++)
 ;
```

In both cases, if the simulation terminates before the condition is fulfilled, other steps are not required to be executed.

6.4 Influencing Other Processes

J-Sim provides two methods in class `JSimProcess` allowing processes to influence other processes present in the system: `cancel()` and `activate()`.

The first one deletes all process' events from the calendar, the second one adds a new event to the calendar.

In order to influence another process, a process needs a reference to it. Since processes have no access to other processes during their life, they must obtain this reference upon their creation and store it into a variable. That is shown in the following example (file `MyProcess.java`):

```
public class MyProcess extends JSimProcess
{
 public JSimProcess anotherProcess;

 public MyProcess(String name, JSimSimulation sim,
                  JSimProcess another)
     throws JSimSimulationAlreadyTerminatedException,
            JSimInvalidParametersException,
            JSimTooManyProcessesException
 {
  super(name, sim);
  anotherProcess = another;
 } // constructor

 protected void life()
 {
  // .....
  anotherProcess.cancel();
  // .....
  anotherProcess.activate(time);
  // .....
 } // life
} // class MyProcess
```

Then, the process can influence another process' scheduling. However, sometimes more than one process needs to be influenced. One possible solution is to pass as many processes as needed to the constructor, for example as an array. Another solution is to add a method to the influencing process that other processes will use to set the process to be influenced. In this case, processes using the method must have a reference to the influencing process in order to be able to invoke the method. This can be arranged in the simple way described above – one parameter in the constructor. Also note that the method can be invoked from outside only when the influencing process is passive, i.e. another process (which calls the method) is active. Therefore,

only one process can be influenced during one step. A simple example is shown below:

```
public class MyProcess extends JSimProcess
{
 public JSimProcess anotherProcess;

 // No changes in constructor
 // No changes in life()

 public void setAnother(JSimProcess another)
 {
  anotherProcess = another;
 } // setAnother
} // class MyProcess
```

6.5 Working with Queues

Before any queue can be created and used, a simulation must exist which will be used as a parameter of queue's constructor. All this is usually done in the main program but not necessarily. An example follows (file `Example.java`):

```
public static void main(String[] args)
{
 JSimSimulation        sim = null;
 JSimHead              queue = null;
 try
 {
  sim = new JSimSimulation("Simulation with queues");
  queue = new JSimHead("Little queue", sim);
  // ... other code
} // main
```

As soon as the queue is created, some elements can be inserted into it. When creating them, the data they hold are passed to them as the only parameter of their constructor. In the following example, five integer numbers will be inserted into the queue already created in previous example:

```
// .....
JSimLink  element;
int       i;
// .....
```

```
for (i = 0; i < 5; i++)
{
 element = new JSimLink(new Integer(i));
 element.into(queue);
} // for
```

Of course, the user is not limited to integer values only. Instances of any class can be inserted into the queue, for example floating-point numbers. In the following example, two floating-point numbers are inserted into the queue, one just after the first element and one just before the last one, using `JSimLink`'s methods `follow()` and `precede()`:

```
// .....
JSimLink  before, after;
// .....
after  = new JSimLink(new Float(0.5));
before = new JSimLink(new Float(3.5));
after.follow(queue.first());
before.precede(queue.last());
```

To retrieve the data back and to print it, the user must distinguish between objects of different data types. Every element's data type can be obtained using its `getDataType()` method. Then, the user can decide whether to retype the data obtained from `getData()` to `Integer` or to `Float`. The following example prints out all data present in the queue, removing them from the queue as soon as they are printed out. To remove an element from the queue, its `out()` method is called. Since there are not usually any other reference to the element, it is disposed automatically by JVM. After each removal, the number of elements left in the queue is printed out.

```
// .....
String    dataType;
Integer   intObject;
Float     floatObject;
// .....
while (!queue.empty())
{
 element = queue.first();
 dataType = element.getDataType();
 System.out.println("The first element is of type: "
                    + dataType);
 switch (dataType.charAt(dataType.lastIndexOf('.') + 1))
```

```
 {
  case 'I': // Integer
   intObject = (Integer) element.getData();
   System.out.println("Data: " + intObject.toString());
   break;
  case 'F': // Float
   floatObject = (Float) element.getData();
   System.out.println("Data: " + floatObject.toString());
   break;
  default:
   System.out.println("There is an unknown object at"
                      + " the head of the queue.");
 } // switch
 element.out();
 System.out.println("Number of items in the queue: "
                    + queue.cardinal());
} // while
```

7 Conclusions

In this paper, many aspects of the J-Sim simulation tool have been discussed: the theoretical background, the kernel and main principles it is based on, the queues, the exception system and many other little details. Finally, some useful practical examples have been presented which will certainly be appreciated mainly by programmers using J-Sim in their everyday life.

The author believes that J-Sim will become at least as popular as C-Sim, its already highly-developed 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. Since detailed API documentation and many ready-to-run examples, having form of a little tutorial, are distributed together with the tool, developing discrete-simulations applications becomes much easier than it has been so far. Finally, J-Sim's graphics possibilities, as well as its abilities to run in any Java environment, make it a candidate for the most used discrete-simulations tool in near future.

J-Sim is available for free at `http://home.zcu.cz/~jkacer/jsim`.

References

[Lea] *Lea, D.*: **Concurrent Programming in Java – Design Principles and Patterns**, Second Edition.
Addison-Wesley, U.S.A., November 2000, ISBN 0-201-31009-0

[Her] *Herout, P.*: **Učebnice jazyka Java**.
Kopp, Czech Republic, June 2000, ISBN 80-7232-115-3

[Rac] *Racek, S. – Roubín, M.*: **Pravděpodobnostní modely počítačů**.
Lecture Notes, University of West Bohemia Publishing, Czech Republic, September 1996, ISBN 80-7082-300-3

[Hlav] *Hlavička, J. - Racek, S. - Herout, P.*: **C-Sim v.4.1**.
Research Report DC-99-09, DCSE CTU Prague Publishing, Czech Republic, December 1999

[Java] **Java Home Page**: `java.sun.com`

[C-Sim] **C-Sim Home Page**: `www.c-sim.zcu.cz`

Index