SSJ User's Guide

Package `simprocs`

Process-driven Simulation Management

Version: September 4, 2007

This package offers some (not-so-efficient) facilities for process-driven simulation. It provides suspend, resume, and synchronization tools for simulated processes. It has two different implementations, one based on Java threads and the other using an interpreter that translates everything to events. The first implementation was developed for *green threads*, which are essentially *simulated* threads and are available in the JDK 1.3.1 and earlier environments by running programs with `java -classic` option. Unfortunately, more recent Java virtual machines offer only native threads. This limits the number of processes, slows down the execution, and sometimes prevent the programs from running properly. The second (slower) implementation uses an interpreter borrowed from the DSOL simulation system to simulate process interactions in a single thread.

Contents

Overview

Process-oriented simulation is managed through this package. A *Process* can be seen as an *active object* whose behavior in time is described by a method called `actions()`. Each process must extend the `SimProcess` class and must implement this `actions()` method. Processes are created and can be scheduled to start at a given simulation time just like events. In contrast with the corresponding `actions()` method of events, the method of processes is generally not executed instantaneously in the simulation time frame. At any given simulation time, at most one process can be *active*, i.e., executing its `actions()` method. The active process may create and schedule new processes, kill suspended processes, and suspend itself. A process is suspended for a fixed delay or until a resource becomes available, or a condition becomes true. When a process is suspended or finishes its execution, another process usually starts or resumes.

These processes may represent "autonomous" objects such as machines and robots in a factory, customers in a retail store, vehicles in a transportation or delivery system, etc. The process-oriented paradigm is a natural way of describing complex systems [**?**, **?**, **?**, **?**] and often leads to more compact code than the event-oriented view. However, it is often preferred to use events only, because this gives a faster simulation program, by avoiding the process-synchronization overhead. Most complex discrete-event systems are quite conveniently modeled only with events. In SSJ, events and processes can be mixed freely. The processes actually use events for their synchronization.

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

Two different implementations of processes are available in SSJ, each one corresponding to a `SimProcess` base class. The first uses Java threads as described in Section 4 of [**?**]. The second is taken from DSOL [**?**, **?**] and was provided to us by Peter Jacobs. Unfortunately, none of these two implementations is fully satisfactory.

Java threads are designed for *real parallelism*, not for the kind of *simulated* parallelism required in process-oriented simulation. In the Java Development Kit (JDK) 1.3.1 and earlier, *green threads* supporting simulated parallelism were available and our original implementation of processes described in [**?**] is based on them. But green threads are no longer supported in recent Java runtime environments. True (native) threads from the operating system are used instead. This adds significant overhead and prevents the use of a large number of processes in the simulation. *This implementation of processes with threads can be used safely only with the JDK versions 1.3.1 or earlier.* A program using the thread-based

process view can easily be 10 to 20 times slower than a similar program using the event view only (see [**?**] for an example).

The second implementation, made by P. Jacobs, stays away from threads. It uses a Java reflection mechanism that interprets the code of processes at runtime and transforms everything into events. All of this is completely transparent to the user. There is no need to change the Java simulation program in any way, except for the `import` statement, at the beginning of the program, that decides which sub-package of `simprocs` we are going to use. Using `simprocs.dsol.SimProcess` instead of `simprocs.SimProcess` will select the DSOL implementation instead of that based on threads. A program using the process view implemented with the DSOL interpreter can be 500 to 1000 times slower than the corresponding event-based program but the number of processes is limited only by the available memory.

Whenever processes are used in a simulation, the simulation clock and the event list should be initialized via `SimProcess.init` instead of `Sim.init`. This kills all processes that may be currently suspended or delayed.

AbstractSimProcess

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

Just like an event, a process must first be constructed, then scheduled. Scheduling a process is equivalent to placing an event in the event list that will start this process when the simulation clock reaches the specified time. The `toString` method can be overridden to return information about the process. This information will be returned when formating the event list as a string, if the process is delayed.

A process can be in one of the following states: `INITIAL`, `EXECUTING`, `DELAYED`, `SUSPENDED`, and `DEAD` (see the diagram). At most *one* process can be in the `EXECUTING` state at any given time, and when there is one, this executing process (called the *current process*) is said to *have control* and is executing one of the instructions of its `actions` method. A process that has been constructed but not yet scheduled is in the `INITIAL` state. A process is in the `DELAYED` state if there is a planned event in the event list to activate it (give it control). When a process is scheduled, it is placed in the `DELAYED` state. It is in the `SUSPENDED` state if it is waiting to be reactivated (i.e., obtain control) without having an event to do so in the event list. A process can suspends itself by calling `suspend` directly or indirectly (e.g., by requesting a busy `Resource`). Usually, a `SUSPENDED` process must be reactivated by another process or event via the `resume` method. A process in the `DEAD` state no longer exists.

It is important to note that when processes are used in a simulation, the simulation clock and event list must be initialized via `SimProcess.init` instead of `Sim.init`.

To construct new processes, the user needs to extend one of the two subclasses of `AbstractSimProcess`. A simulation program should always use the same process base class. The `SimProcess` subclass implements processes by using Java threads. This is the fastest implementation. The `SimProcess` from package `simprocs.dsol` uses the DSOL interpreter to simulate processes in a single-threaded program, but it is much slower than the multi-threaded implementation.

```
package umontreal.iro.lecuyer.simprocs;

public abstract class AbstractSimProcess
```

Possible states of a process

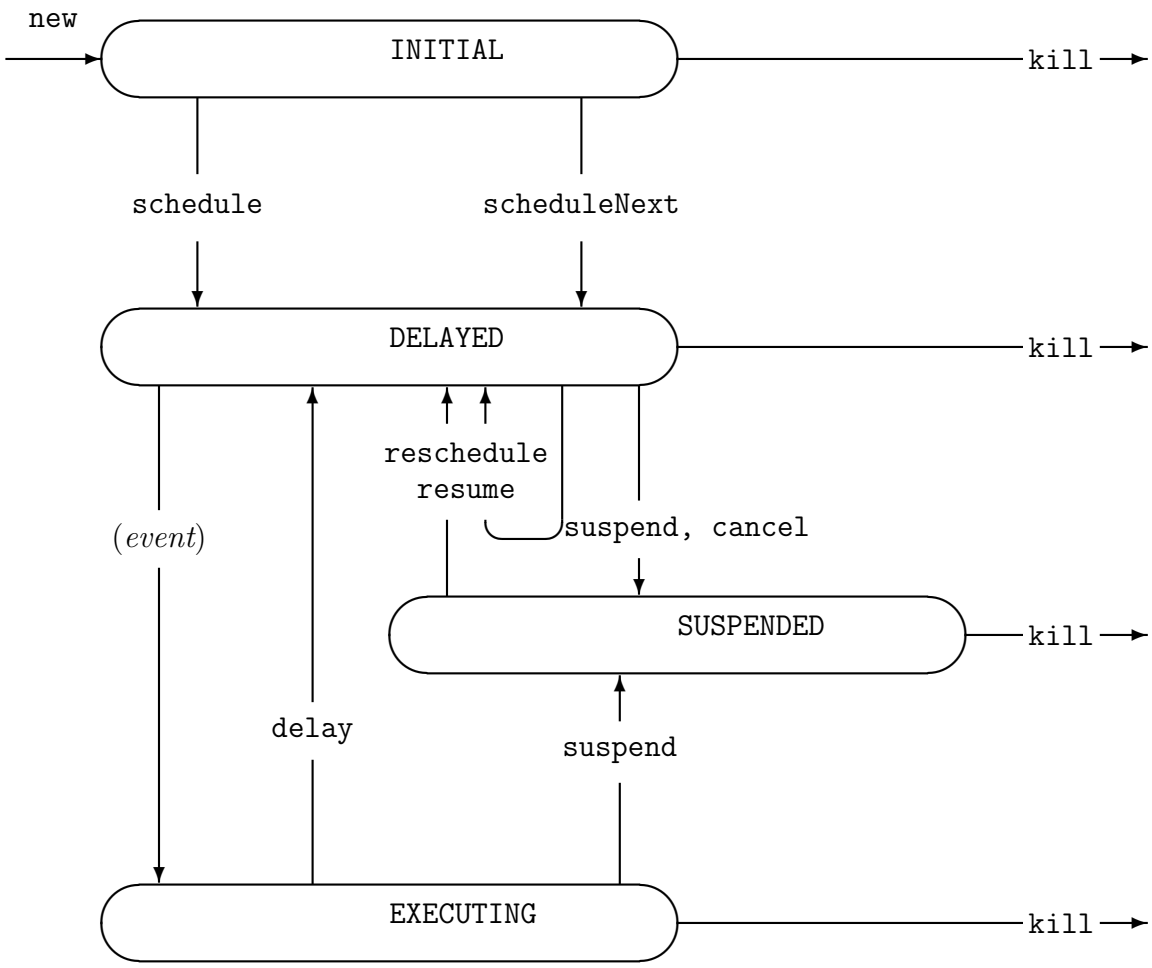
```
public static final int INITIAL   = 0;
```

The process has been created but not yet scheduled.

```
public static final int EXECUTING = 1;
```

The process is the one currently executing its `actions` method.

Figure 1: The possible states and state transitions of a `SimProcess`



`public static final int DELAYED = 2;`

The process is not executing but has an event in the event list to reactivate it later on.

`public static final int SUSPENDED = 3;`

The process is not executing and will have to be reactivated by another process or event later on.

`public static final int DEAD = 4;`

The process has terminated its execution.

Constructor

`public AbstractSimProcess()`

Constructs a new process without scheduling it. It will have to be scheduled later on. The

process is in the `INITIAL` state.

Methods

`public void schedule (double delay)`

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

`public void scheduleNext()`

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

`public static AbstractSimProcess currentProcess()`

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

`public final boolean isAlive()`

Returns `true` if the process is alive, otherwise `false`.

`public int getState()`

Returns the state of the process.

`public double getDelay()`

If the process is in the `DELAYED` state, returns the remaining time until the planned occurrence of its activating event. Otherwise, an illegal state exception will be thrown printing an error message.

`public abstract void delay (double delay);`

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

`public void reschedule (double delay)`

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

`public abstract void suspend();`

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

`public void resume()`

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

`public boolean cancel()`

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

`public abstract void kill();`

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

`public abstract void actions();`

This is the method that is called when this process is executing. Every subclass of `SimProcess` that is to be instantiated must provide an implementation of this method.

SimProcess

Represents a simulation process with an associated Java thread. The simulation process threads are synchronized so only one process runs at a time.

Note: the user needs to ensure that the `actions` method of any process can be terminated, i.e., no infinite loops. If such a method never terminates, threads will not be recycled, causing memory problems.

```
package umontreal.iro.lecuyer.simprocs;

public abstract class SimProcess extends AbstractSimProcess
```

Constructor

`public SimProcess()`

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

Methods

`public static void init()`

Initializes the process-driven simulation. This kills all processes and calls `Sim.init`.

`public static void init (EventList evlist)`

Initializes the simulation and sets the given event list `evlist` to be used by the simulation executive.

`public static void killAll()`

Kills all instances of the class `SimProcess`.

Note: this method is unstable under Java versions 1.4 or greater.

dsol.SimProcess

Represents a simulation process whose `actions` method is interpreted by the DSOL interpreter [**?**], written by Peter Jacobs (`http://www.tbm.tudelft.nl/webstaf/peterja/index.htm`). When a process executes, a virtual machine implemented in Java is invoked to interpret the byte-code. The processes are then simulated in a single Java thread, which allows a much larger number of threads than when each process has its own native thread, at the expense of a slower execution time. To use this implementation in a program instead of the thread-based implementation provides in the standard `SimProcess` class, it suffices to replace the import statement

```
import umontreal.iro.lecuyer.simprocs.SimProcess;
```

by

```
import umontreal.iro.lecuyer.simprocs.dsol.SimProcess;
```

```
package umontreal.iro.lecuyer.simprocs.dsol;

public abstract class SimProcess extends AbstractSimProcess
```

Constructor

`public SimProcess()`

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

Methods

`public static void init()`

Initializes the process-driven simulation. This kills all processes and calls `Sim.init`.

`public static void init (EventList evlist)`

Initializes the simulation and sets the given event list `evlist` to be used by the simulation executive.

Resource

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

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

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

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

```
package umontreal.iro.lecuyer.simprocs;

public class Resource
```

Constructors

`public Resource (int capacity)`

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

`public Resource (int capacity, String name)`

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

Methods

`public void setStatCollecting (boolean b)`

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

`public void initStat()`

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

`public void init()`

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

`public int getCapacity()`

Returns the current capacity of the resource.

`public void setPolicyFIFO()`

Set the service policy to FIFO (first in, first out): the processes are placed in the list (and served) according to their order of arrival.

`public void setPolicyLIFO()`

Set the service policy to LIFO (last in, first out): the processes are placed in the list (and served) according to their inverse order of arrival, the last arrived are served first.

`public void changeCapacity (int diff)`

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

`public void setCapacity (int newcap)`

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

`public int getAvailable()`

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

`public void request (int n)`

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

`public void release (int n)`

The executing process that invokes this method releases `n` units of the resource. If this process is holding exactly `n` units, it is removed from the service list of this resource and its `UserRecord` object disappears. If this process is holding less than `n` units, the program throws an illegal argument exception. If there are other processes waiting for this resource whose requests can now be satisfied, they obtain the resource.

`public LinkedListStat waitList()`

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

`public LinkedListStat servList()`

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

`public Accumulate statOnCapacity()`

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

`public Accumulate statOnUtil()`

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

`public Tally statOnSojourn()`

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

`public String getName()`

Returns the name (or identifier) associated to this resource. If it was not given upon resource construction, this returns `null`.

`public String report()`

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

Bin

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

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

```
package umontreal.iro.lecuyer.simprocs;

public class Bin
```

Constructors

`public Bin()`

Constructs a new bin, initially empty, with service policy FIFO.

`public Bin (String name)`

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

Methods

`public void init()`

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

`public void setStatCollecting (boolean b)`

Starts or stops collecting statistics on the list returned by `waitList` for this bin. If the statistical collection is turned ON, It also constructs (if not yet done) and initializes an additional statistical collector of class `Accumulate` for this bin. This collector will be updated automatically. It can be accessed via `statOnAvail`, and monitors the evolution of the available tokens of the bin as a function of time.

`public void initStat()`

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

`public void setPolicyFIFO()`

Sets the service policy for ordering processes waiting for tokens on the bin to FIFO (first in, first out): the processes are placed in the list (and served) according to their order of arrival.

`public void setPolicyLIFO()`

Sets the service policy for ordering processes waiting for tokens on the bin to LIFO (last in, first out): the processes are placed in the list (and served) according to their inverse order of arrival, the last arrived are served first.

`public int getAvailable()`

Returns the number of available tokens for this bin.

`public void take (int n)`

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

`public void put (int n)`

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

`public LinkedListStat waitList()`

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

`public Accumulate statOnAvail()`

Returns the statistical collector for the available tokens on the bin as a function of time. This collector exists only if `setStatCollecting (true)` has been invoked previously.

`public String report()`

Returns a string containing a complete statistical report on this bin. The method `setStatCollecting (true)` must have been invoked previously, otherwise no statistics will have been collected. The report contains statistics on the available tokens, queue size and waiting time for this bin.

Condition

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

```
package umontreal.iro.lecuyer.simprocs;
```

```
public class Condition extends Observable
```

Constructors

`public Condition (boolean val)`

Constructs a new `Condition` with initial value `val`.

`public Condition (boolean val, String name)`

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

Methods

`public void init (boolean val)`

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

`public void set (boolean val)`

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

`public boolean state()`

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

`public void waitFor()`

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

`public LinkedListStat waitList()`

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

`public String getName()`

Returns the name (or identifier) associated to this condition.

`public void setBroadcasting (boolean b)`

Instructs the condition to start or stop observation broadcasting. When this is turned ON, a `Boolean` observation is notified to registered observers when the state of the condition changes. This boolean gives the new state of the condition.

Warning: Since this can reduce program performance, this should be turned on only when there are registered observers.

UserRecord

This class represents a record object to store information related to the request of a process for a `Resource` or for `Bin` tokens, or when a process waits for a `Condition`. A `UserRecord` is created for each process request. The record contains the number of units requested or used, the associated process, and the simulation time when the request was made. Lists of processes waiting for a `Resource`, `Bin`, or `Condition`, for example, contain `UserRecord` objects.

```
package umontreal.iro.lecuyer.simprocs;
```

```
public class UserRecord
```

`public int getNumUnits()`

Returns the number of units requested or used by the associated process.

`public AbstractSimProcess getProcess()`

Returns the process object associated with this record.

`public double getRequestTime()`

Return the time of creation of this record.

References

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

[2] W. R. Franta. *The Process View of Simulation*. North Holland, New York, 1977.

[3] P. H. M. Jacobs. *DSOL: an open source, Java based, suite for continuous and discrete event simulation*. Technische Universiteit Delft, Delft, Netherlands, 2005. Available at `http://www.simulation.tudelft.nl/dsol/`.

[4] P. H. M. Jacobs and A. Verbraeck. Single-threaded specification of process-interaction formalism in Java. In R. G. Ingalls, M. D. Rosetti, J. S. Smith, and B. A. Peters, editors, *Proceedings of the 2004 Winter Simulation Conference*, pages 1548–1555. IEEE Press, 2004.

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

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

[7] P. L'Ecuyer and E. Buist. Simulation in Java with SSJ. In *Proceedings of the 2005 Winter Simulation Conference*, pages 611–620. IEEE Press, 2005.

[8] P. L'Ecuyer, L. Meliani, and J. Vaucher. SSJ: A framework for stochastic simulation in Java. In E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, editors, *Proceedings of the 2002 Winter Simulation Conference*, pages 234–242. IEEE Press, 2002.