

# Introduction to SimPy Internals

Norm Matloff

February 20, 2008  
©2006-8, N.S. Matloff

## Contents

<b>1 Purpose</b>	<b>1</b>
<b>2 Python Generators</b>	<b>2</b>
<b>3 How SimPy Works</b>	<b>4</b>
3.1 Running Example . . . . .	4
3.2 How <code>initialize()</code> Works . . . . .	4
3.3 How <code>activate()</code> Works . . . . .	4
3.4 How <code>simulate()</code> Works . . . . .	5
3.4.1 The Core while Loop . . . . .	5
3.4.2 Call to <code>_nextev()</code> . . . . .	5
3.4.3 How a New Event Gets Added to the Event List . . . . .	7
3.5 How <code>Resource()</code> , <code>yield request</code> and <code>yield release</code> Work . . . . .	7
<b>A SimPy Source Code</b>	<b>9</b>

## 1 Purpose

In simulation (and other) languages, one often wonders “What does this operation REALLY do?” The description in the documentation may not be fully clear, say concerning the behavior of the operation in certain specialized situations. But in the case of open source software like SimPy, we can actually go into the code to see what the operation really does.

Another reason why access to the language’s internals is often useful is that it can aid our debugging activities. We can check the values of the internal data structures, and so on.

Accordingly, this unit will be devoted to introducing the basics of SimPy internals. We will use SimPy version 1.9 as our example.

## 2 Python Generators

SimPy is built around Python **generators**, which are special kinds of Python functions. Following will be a quick overview of generators, sufficient for our purposes here. If you wish to learn more about generators, see the generators unit in my Python tutorial, at my Python tutorials Web site, <http://heather.cs.ucdavis.edu/~matloff/python.html>.

Speaking very roughly in terms of usage, a generator is a function that we wish to call repeatedly, but which is unlike an ordinary function in that successive calls to a generator function don't start execution at the beginning of the function. Instead, the current call to a generator function will resume execution right after the spot in the code at which the last call exited, i.e. we "pick up where we left off."

Here is a concrete example:

```
1 # yieldex.py example of yield, return in generator functions
2
3 def gy():
4     x = 2
5     y = 3
6     yield x,y,x+y
7     z = 12
8     yield z/x
9     print z/y
10    return
11
12 def main():
13     g = gy()
14     print g.next() # prints x, y and x+y
15     print g.next() # prints z/x
16     print g.next() # causes the exception
17
18 if __name__ == '__main__':
19     main()
```

```
1 % python yieldex.py
2 (2, 3, 5)
3 6
4 4
5 Traceback (most recent call last):
6   File "yieldex.py", line 19, in ?
7     main()
8   File "yieldex.py", line 16, in main
9     print g.next()
10 StopIteration
```

Here is what happened in the execution of that program:

- As with any Python program, the Python interpreter started execution at the top of the file. When the interpreter sees free-standing code, it executes that code, but if it encounters a function definition, it records it. In particular, the interpreter notices that the function **gy()** contains a **yield** statement, and thus records that this function is a generator rather than an ordinary function. Note carefully that the function has NOT been executed yet at this point.

- The line

```
g = gy()
```

creates a Python **iterator**, assigning it to **g**. Again, to learn the details on iterators, you can read my tutorial above, but all you need to know is that **g** is a certain kind of object which includes a member function named **next()**, and that this function will be our vehicle through which to call **gy()**. Note carefully that **gy()** STILL has not been executed yet at this point.

- The three statements

```
print g.next()
print g.next()
print g.next()
```

call **gy()** three times, in each case printing out the value returned by that function, either through **yield** or the traditional **return**.

- With the first call, only the lines

```
x = 2
y = 3
yield x, y, x+y
```

are executed. The **yield** acts somewhat like a classical return, in the sense that (a) control passes back to the caller, in this case **main()**, and (b) a value is returned, in this case the tuple **(x,y,x+y)**.<sup>1</sup> This results in (2,3,5) being printed out.

But the difference between **yield** and **return** is that **yield** also records the point at which we left the generator. In this case here, it means that it will be recorded that our **yield** operation was executed at the first of the two **yield** statements in **gy()**.

- The second call to **g.next()** in **main()** will therefore begin right after the last **yield**, meaning that this second call will begin at the line

```
z = 12
```

instead of at the line

```
x = 2
```

Moreover, the values of the local variables, here **x** and **y**,<sup>2</sup> will be retained; for instance, **y** will still be 3.

- Execution will then proceed through the next **yield**,

```
yield z/x
```

This again will return control to the caller, **main()**, along with the return value **z/x**. Again, it will be noted that the **yield** which executed this time was the second **yield**.

- The third call to **g.next()** causes an execution error. It is treated as an error because a call to a **next()** function for a generator assumes that another **yield** will be encountered, which wasn't the case here. We could have our code sense for this **StopIteration** condition by using Python's **try** construct.

---

<sup>1</sup>Recall that the parentheses in a tuple are optional if no ambiguity would result from omitting them.

<sup>2</sup>The local **z** has not come into existence yet.

## 3 How SimPy Works

Armed with our knowledge of generators, we can now take a look inside of SimPy. I've included the source code, consisting of the file **Simulation.py** for version 1.9 of SimPy, in an appendix to this document.

### 3.1 Running Example

Here and below, let's suppose we have a class in our application code named **X**, which is a subclass of **Process**, and whose PEM is named **Run()**, and that we have created an instance of **X** named **XInst**.

The key point to note is that since **Run()** contains one or more **yield** statements, the Python interpreter recognizes it as a generator. Thus the call **XInst.Run()** within our call to **activate()** (see below) returns an iterator. I'll refer to this iterator here as **XIt** for convenience, though you'll see presently that the SimPy code refers to it in another way. But the point is that **XIt** will be our thread.

### 3.2 How initialize() Works

This function does surprisingly little. Its main actions are to set the global variables **\_t**, **\_e** and **\_stop**, which play the following roles:

- The global **\_t** stores the simulated time, initialized to 0. (The application API **now()** simply returns **\_t**.)
- The global **\_e** is an instance of the class **\_Evlist**. One of the member variables of that class is **events**, which is the event list.
- The global **\_stop** is a flag to stop the simulation. For example, it is set when **stopSimulation()** is called.

### 3.3 How activate() Works

What happens when our application code executes the following line?

```
activate(XInst,XInst.Run())
```

The definition of **activate()** begins with

```
def activate(obj,process,at="undefined",delay="undefined",prior=False):
```

so in our call

```
activate(XInst,XInst.Run())
```

the formal parameter **obj** will be **XInst**, an instance of a subclass of **Process**, and **process** will be our iterator **XIt**. (As you can see, we have not used the optional named parameters here.)

At this point **activate()** executes its code

```
obj._nextpoint=process
```

Recall that our class **X** is a subclass of SimPy's **Process**. One of the member variables of the latter is **\_nextpoint**, and you can now see that it will be our iterator, i.e. our thread. The name of this member variable alludes to the fact that each successive call to a generator “picks up where we last left off.” The variable's name can thus be thought of as an abbreviation for “point at which to execute next.”

Finally, **activate()** sets **zeit** to the current simulated time **\_t**. (The more general usage of **activate()** allows starting a thread later than the current time, but let's keep things simple here.)

Then **activate()** executes

```
_e._post(obj,at=zeit,prior=prior)
```

Here is what that does: Recall that **\_e** is the object of class **\_Evlist**, which contains our event list. A member function in that class is **\_post()**, whose role is to add (“post”) an event to the event list. In our case here, there is no real event, but the code will add an artificial event for this thread. The time for this artificial event will be the current time. The effect of this will be that the first execution of this thread will occur “immediately,” meaning at the current simulated time. This is what gets the ball rolling for this thread.

## 3.4 How `simulate()` Works

### 3.4.1 The Core `while` Loop

The core of **simulate()** consists of a **while** loop which begins with

```
while not _stop and _t<=_endtime:
```

Here **\_endtime** is the maximum simulated time set by the application code, and you'll recall that **\_stop** is a flag that tells SimPy to stop the simulation.

In each iteration of this **while** loop, the code pulls the event with the earliest simulated time from the event list, updates the current simulated time to that time, and then calls the iterator associated with that event. Remember, that iterator is our thread, so calling it will cause the thread to resume execution. You will see more details in the next section.

### 3.4.2 Call to `_nextev()`

A key statement near the top of the core **while** loop of **simulate()** is

```
a=nextev()
```

Here **nextev** is an alternate name the authors of SimPy gave to a member function of the **\_Evlist** class, **\_nextev()**.

The function **\_nextev()** extracts the next event, acts on it (e.g. updating the simulated clock), and then has the event's associated thread resume execution until it next hits a **yield**. The latter causes a return

to the caller. That returned value consists of a tuple that in the case of our example class **X** above will be **(yield\_value,XInst)**, where **yield\_value** is the tuple returned by the thread. Following are some of the details.

This version of SimPy stores the events in a heap, using the Python library **heapq**. The latter stores a heap in a Python list, which in our case here is the member variable **timestamps** in the **\_Evlist** class. Here is the line within **\_nextev()** that extracts the earliest event:

```
(_tnotice, p,nextEvent,cancelled) = heapq.heappop(self.timestamps)
```

That variable **\_tnotice** now contains the time for this event. The function then updates the simulated time to that time, and checks to see whether the simulation's specified duration has been reached:

```
_t = _tnotice
if _t > _endtime:
    _t = _endtime
    _stop = True
```

Eventually this function **\_nextev()** executes the statement

```
resultTuple = nextEvent._nextpoint.next()
```

Again, recall that **\_nextpoint** is the iterator for this thread. Thus this statement will call the iterator, which causes the thread to resume execution. As noted above, the thread will eventually encounter a **yield**, returning execution to the above statement, and assigning to **resultTuple** the value returned by the **yield**.

Let's recall what **resultTuple** will look like. For instance the statement

```
yield hold,self,0.6
```

returns the 3-tuple **(hold,self,0.6)**, where **hold** is a numerical code, from a set defined in **Simulation.py**:

```
# yield keywords
hold=1
passivate=2
request=3
release=4
waitevent=5
queueevent=6
waituntil=7
get=8
put=9
```

Finally **\_nextev()** executes

```
return (resultTuple, nextEvent)
```

where as mentioned, **nextEvent** is our **Process** instance, e.g. **XInst** in our example above. Note that at this point, we have started to set up the next event for this thread, in the information contained in that return tuple. Now we must add it to the event list.

### 3.4.3 How a New Event Gets Added to the Event List

After calling and performing some checks, `_nextev()`, `simulate()` then executes

```
command = a[0][0]
dispatch[command](a)
```

Here's how what happens: Recall that `a` is the object returned by our call to `_nextev()` that we extracted from the event list, and that *inter alia* it contains the tuple returned when this thread last executed a **yield**. The first element of that tuple will be one of **hold**, **request** etc. This is the basis for formulating our next event, as follows.

SimPy defines a Python dictionary **dispatch** of functions, which serves as a lookup table:

```
dispatch={hold:holdfunc, request:requestfunc, release:releasefunc, \
          passivate:passivatefunc, waitevent:waitevfunc, queueevent:queueevfunc, \
          waituntil:waituntilfunc, get:getfunc, put:putfunc}
```

So, the code

```
command = a[0][0]
dispatch[command](a)
```

has the effect of calling **holdfunc** in the case of **yield hold**, **requestfunc** in the case of **yield request** and so on. Those functions in turn calls others that do the real work. For instance, **holdfunc()** in turn calls `_hold()`, which does

```
_e._post(what=who, at=_t+delay)
```

As you recall, the function `_post()` adds a new event to the event list. The argument **who** here is our event, say **XInst**, and **delay** is the time that **XInst.Run()** asked to hold in its **yield hold** statement, say 0.6. So, you can see that the code above is scheduling an event 0.6 amount of time from now, which is exactly what we want. **XInst's nextTime** field (inherited from the **Process** class) will then be set to `_t+delay`

The function `_post()` adds this new event to the event list, via its line

```
hq.heappush(self.timestamps, what._rec)
```

As mentioned, the heap `_e.timestamps` is a Python list, consisting of instances of **Process** subclasses, i.e. consisting of threads. So, we're adding our new event, **what.\_rec**, to the events heap.

### 3.5 How Resource(), yield request and yield release Work

Suppose our application code also sets up some resources:

```
R = Resource(2)
```

Recall that **Resource** is a SimPy class, so here we are calling that class' constructor with an argument of 2, meaning that we want two servers or machines or whatever. The constructor includes code

```
self.capacity=capacity # resource units in this resource
...
self.n=capacity # uncommitted resource units
```

The formal parameter **capacity** has the actual value 2 in our example here, and as you can see, it is now stored in a member variable of **Process** of the same name. Furthermore, the member variable **n**, which stores the current number of free units of the resource, is initially set to the capacity, i.e. all units are assumed available at the outset.

At this time, the constructor also sets up two other member variables (and more we aren't covering here):

- **waitQ**, the queue of jobs waiting to a unit of this resource
- **activeQ**, the list of jobs currently using a unit of this resource

For **yield request**, **simulate()** calls the function **\_request()**. The key code there is, for the non-preemption case,

```
if self.n == 0:
    self.waitQ.enter(obj)
    # passivate queuing process
    obj._nextTime=None
else:
    self.n -= 1
    self.activeQ.enter(obj)
    _e._post(obj, at=_t, prior=1)
```

As you can see, if there are no available units, we add the thread to the queue for this resource, and passivate the thread. But if there is an available unit, the code creates an artificial event, to be executed immediately (as with **activate()**, this is immediate in the sense of being at the same simulated time), and adds it to the event list.

Note that the way that passivation is done is to simply set the thread's **nextTime** field (time of the next event for this thread) to None. This is the way **yield passivate** is handled too:

```
def _passivate(self, a):
    a[0][1]._nextTime=None
```

On the other hand, if there are units available, we grab one, thus decrementing **n** by 1, add the thread to the list of threads currently using the units, and then add this thread to the event list. Since its event time will be **now()**, it will start right back up again immediately in the sense of simulated time, though it may not be the next thread to run.

When a **yield release** statement is executed by the application code, the natural actions are then taken by the function **\_release()**:

```
self.n += 1
self.activeQ.remove(arg[1])
```



```

#reactivate first waiting requestor if any; assign Resource to it
if self.waitQ:
    obj=self.waitQ.leave()
    self.n -= 1 #assign 1 resource unit to object
    self.activeQ.enter(obj)
    reactivate(obj,delay=0,prior=1)

```

(Here again I've omitted code, e.g. for the pre-emptable case, to simplify the exposition.)

## A SimPy Source Code

Below is the SimPy source code. I've removed some of the triple-quoted comments at the beginning, and the test code at the end.

```

1  #!/usr/bin/env python
2  from SimPy.Listener import *
3  import heapq as hq
4  import types
5  import sys
6  import new
7  import random
8  import inspect
9
10 # $Revision: 1.1.1.75 $ $Date: 2007/12/18 13:30:47 $ kgm
11 """Simulation 1.9 Implements SimPy Processes, Resources, Buffers, and the backbone simulation
12 scheduling by coroutine calls. Provides data collection through classes
13 Monitor and Tally.
14 Based on generators (Python 2.3 and later)
15 """
16
17 # yield keywords
18 hold=1
19 passivate=2
20 request=3
21 release=4
22 waitevent=5
23 queueevent=6
24 waituntil=7
25 get=8
26 put=9
27
28 _endtime=0
29 _t=0
30 _e=None
31 _stop=True
32 _wustep=False #controls per event stepping for waituntil construct; not user API
33 try:
34     True, False
35 except NameError:
36     True, False = (1 == 1), (0 == 1)
37 condQ=[]
38 allMonitors=[]
39 allTallies=[]
40
41 def initialize():
42     global _e,_t,_stop,condQ,allMonitors,allTallies
43     _e=__Evlist()
44     _t=0
45     _stop=False
46     condQ=[]
47     allMonitors=[]
48     allTallies=[]
49
50 def now():
51     return _t
52
53 def stopSimulation():
54     """Application function to stop simulation run"""
55     global _stop
56     _stop=True
57
58 def _startWUStepping():
59     """Application function to start stepping through simulation for waituntil construct."""
60     global _wustep
61     _wustep=True
62
63 def _stopWUStepping():
64     """Application function to stop stepping through simulation."""
65     global _wustep

```

```

66     _wustep=False
67
68 class Simerror(Exception):
69     def __init__(self,value):
70         self.value=value
71
72     def __str__(self):
73         return `self.value`
74
75 class FatalSimerror(Simerror):
76     def __init__(self,value):
77         Simerror.__init__(self,value)
78         self.value=value
79
80 class Process(Lister):
81     """Superclass of classes which may use generator functions"""
82     def __init__(self,name="a_process"):
83         #the reference to this Process instances single process (==generator)
84         self.nextpoint=None
85         self.name=name
86         self.nextTime=None #next activation time
87         self._remainService=0
88         self._preempted=0
89         self._priority={}
90         self._getpriority={}
91         self._putpriority={}
92         self._terminated= False
93         self._inInterrupt= False
94         self.eventsFired=[] #which events process waited/queued for occurred
95
96     def active(self):
97         return self._nextTime <> None and not self._inInterrupt
98
99     def passive(self):
100         return self._nextTime is None and not self._terminated
101
102     def terminated(self):
103         return self._terminated
104
105     def interrupted(self):
106         return self._inInterrupt and not self._terminated
107
108     def queuing(self,resource):
109         return self in resource.waitQ
110
111     def cancel(self,victim):
112         """Application function to cancel all event notices for this Process
113         instance;(should be all event notices for the _generator_)."""
114         _e._unpost (whom=victim)
115
116     def start(self,pem=None,at="undefined",delay="undefined",prior=False):
117         """Activates PEM of this Process.
118         p.start (p.pemname ([args]) [, (at= t |delay=period)][,prior=False] ) or
119         p.start ([p.ACTIONS ()] [, (at= t |delay=period)][,prior=False]) (ACTIONS
120         parameter optional)
121         """
122         if pem is None:
123             try:
124                 pem=self.ACTIONS()
125             except AttributeError:
126                 raise FatalSimerror\
127                     ("Fatal SimPy error: no generator function to activate")
128         else:
129             pass
130         if _e is None:
131             raise FatalSimerror\
132                 ("Fatal SimPy error: simulation is not initialized"\
133                  "(call initialize() first)")
134         if not (type(pem) == types.GeneratorType):
135             raise FatalSimerror("Fatal SimPy error: activating function which"+
136                                " is not a generator (contains no 'yield')")
137         if not self._terminated and not self._nextTime:
138             #store generator reference in object; needed for reactivation
139             self._nextpoint=pem
140             if at=="undefined":
141                 at=t
142             if delay=="undefined":
143                 zeit=max(_t,at)
144             else:
145                 zeit=max(_t,_t+delay)
146             _e._post (what=self,at=zeit,prior=prior)
147
148     def _hold(self,a):
149         if len(a[0]) == 3:
150             delay=abs (a[0][2])
151         else:
152             delay=0
153         who=a[1]
154         self.interruptLeft=delay
155         self._inInterrupt=False
156         self.interruptCause=None
157         _e._post (what=who,at=_t+delay)

```

```

158
159 def _passivate(self,a):
160     a[0][1]._nextTime=None
161
162 def interrupt(self,victim):
163     """Application function to interrupt active processes"""
164     # can't interrupt terminated/passive/interrupted process
165     if victim.active():
166         victim.interruptCause=self # self causes interrupt
167         left=victim._nextTime-t
168         victim.interruptLeft=left # time left in current 'hold'
169         victim._inInterrupt=True
170         reactivate(victim)
171         return left
172     else: #victim not active -- can't interrupt
173         return None
174
175 def interruptReset(self):
176     """
177     Application function for an interrupt victim to get out of
178     'interrupted' state.
179     """
180     self._inInterrupt= False
181
182 def acquired(self,res):
183     """Multi-functional test for renegeing for 'request' and 'get':
184     (1)If res of type Resource:
185         Tests whether resource res was acquired when proces reactivated.
186         If yes, the parallel wakeup process is killed.
187         If not, process is removed from res.waitQ (renegeing).
188     (2)If res of type Store:
189         Tests whether item(s) gotten from Store res.
190         If yes, the parallel wakeup process is killed.
191         If no, process is removed from res.getQ
192     (3)If res of type Level:
193         Tests whether units gotten from Level res.
194         If yes, the parallel wakeup process is killed.
195         If no, process is removed from res.getQ.
196     """
197     if isinstance(res,Resource):
198         test=self in res.activeQ
199         if test:
200             self.cancel(self._holder)
201         else:
202             res.waitQ.remove(self)
203             if res.monitored:
204                 res.waitMon.observe(len(res.waitQ),t=now())
205         return test
206     elif isinstance(res,Store):
207         test=len(self.got)
208         if test:
209             self.cancel(self._holder)
210         else:
211             res.getQ.remove(self)
212             if res.monitored:
213                 res.getQMon.observe(len(res.getQ),t=now())
214         return test
215     elif isinstance(res,Level):
216         test=not (self.got is None)
217         if test:
218             self.cancel(self._holder)
219         else:
220             res.getQ.remove(self)
221             if res.monitored:
222                 res.getQMon.observe(len(res.getQ),t=now())
223         return test
224
225 def stored(self,buffer):
226     """Test for renegeing for 'yield put . . .' compound statement (Level and
227     Store. Returns True if not renegeed.
228     If self not in buffer.putQ, kill wakeup process, else take self out of
229     buffer.putQ (renegeed)"""
230     test=self in buffer.putQ
231     if test: #renegeed
232         buffer.putQ.remove(self)
233         if buffer.monitored:
234             buffer.putQMon.observe(len(buffer.putQ),t=now())
235     else:
236         self.cancel(self._holder)
237     return not test
238
239 def allEventNotices():
240     """Returns string with eventlist as;
241     t1: processname,processname2
242     t2: processname4,processname5, . . .
243     . . . .
244     """
245     ret=""
246     tempList=[]
247     tempList[:]=_e.timestamps
248     tempList.sort()
249     # return only event notices which are not cancelled

```

```

250     tempList=[[x[0],x[2].name] for x in tempList if not x[3]]
251     tprev=-1
252     for t in tempList:
253         # if new time, new line
254         if t[0]==tprev:
255             # continue line
256             ret+=",%s"%t[1]
257         else:
258             # new time
259             if tprev==-1:
260                 ret="%s: %s"%(t[0],t[1])
261             else:
262                 ret+="\n%s: %s"%(t[0],t[1])
263             tprev=t[0]
264     return ret+"\n"
265
266 def allEventTimes():
267     """Returns list of all times for which events are scheduled.
268     """
269     r=[]
270     r[:]=_e.timestamps
271     r.sort()
272     # return only event times of not cancelled event notices
273     rl=[x[0] for x in r if not r[3]]
274     tprev=-1
275     ret=[]
276     for t in rl:
277         if t==tprev:
278             #skip time, already in list
279             pass
280         else:
281             ret.append(t)
282             tprev=t
283     return ret
284
285 class __Evlist(object):
286     """Defines event list and operations on it"""
287     def __init__(self):
288         # always sorted list of events (sorted by time, priority)
289         # make heapq
290         self.timestamps = []
291         self.sortpr=0
292
293     def _post(self, what, at, prior=False):
294         """Post an event notice for process what for time at"""
295         # event notices are Process instances
296         if at < _t:
297             raise Simerror("Attempt to schedule event in the past")
298         what._nextTime = at
299         self.sortpr-=1
300         if prior:
301             # before all other event notices at this time
302             # heappush with highest priority value so far (negative of monotonely increasing number)
303             # store event instance in process instance
304             what._rec=[at,self.sortpr,what,False]
305             # make event list refer to it
306             hq.heappush(self.timestamps,what._rec)
307         else:
308             # heappush with lowest priority
309             # store event notice in process instance
310             what._rec=[at,-self.sortpr,what,False]
311             # make event list refer to it
312             hq.heappush(self.timestamps,what._rec)
313
314     def _unpost(self, whom):
315         """
316         Mark event notice for whom as cancelled if whom is a suspended process
317         """
318         if whom._nextTime is not None: # check if whom was actually active
319             whom._rec[3]=True ## Mark as cancelled
320             whom._nextTime=None
321
322     def _nextev(self):
323         """Retrieve next event from event list"""
324         global _t, _stop
325         noActiveNotice=True
326         ## Find next event notice which is not marked cancelled
327         while noActiveNotice:
328             if self.timestamps:
329                 ## ignore priority value
330                 (_tnotice, p,nextEvent,cancelled) = hq.heappop(self.timestamps)
331                 noActiveNotice=cancelled
332             else:
333                 raise Simerror("No more events at time %s" % _t)
334         _t=_tnotice
335         if _t > _endtime:
336             _t = _endtime
337             _stop = True
338             return (None,)
339         try:
340             resultTuple = nextEvent._nextpoint.next()
341         except StopIteration:

```

```

342         nextEvent._nextpoint = None
343         nextEvent._terminated = True
344         nextEvent._nextTime = None
345         resultTuple = None
346         return (resultTuple, nextEvent)
347
348     def _isEmpty(self):
349         return not self.timestamps
350
351     def _allEventNotices(self):
352         """Returns string with eventlist as
353            t1: [procname,procname2]
354            t2: [procname4,procname5, . . . ]
355            . . . .
356         """
357         ret=""
358         for t in self.timestamps:
359             ret+="s:%s\n"%(t[1]._nextTime, t[1].name)
360         return ret[:-1]
361
362     def _allEventTimes(self):
363         """Returns list of all times for which events are scheduled.
364         """
365         return self.timestamps
366
367
368 def activate(obj,process,at="undefined",delay="undefined",prior=False):
369     """Application function to activate passive process."""
370     if _e is None:
371         raise FatalSimerror\
372             ("Fatal error: simulation is not initialized (call initialize() first)")
373     if not (type(process) == types.GeneratorType):
374         raise FatalSimerror("Activating function which"+
375             " is not a generator (contains no 'yield'")
376     if not obj._terminated and not obj._nextTime:
377         #store generator reference in object; needed for reactivation
378         obj._nextpoint=process
379         if at=="undefined":
380             at=_t
381         if delay=="undefined":
382             zeit=max(_t,at)
383         else:
384             zeit=max(_t,_t+delay)
385         _e._post(obj,at=zeit,prior=prior)
386
387 def reactivate(obj,at="undefined",delay="undefined",prior=False):
388     """Application function to reactivate a process which is active,
389     suspended or passive."""
390     # Object may be active, suspended or passive
391     if not obj._terminated:
392         a=Process("SimPysystem")
393         a.cancel(obj)
394         # object now passive
395         if at=="undefined":
396             at=_t
397         if delay=="undefined":
398             zeit=max(_t,at)
399         else:
400             zeit=max(_t,_t+delay)
401         _e._post(obj,at=zeit,prior=prior)
402
403 class Histogram(list):
404     """ A histogram gathering and sampling class"""
405
406     def __init__(self,name = '',low=0.0,high=100.0,nbins=10):
407         list.__init__(self)
408         self.name = name
409         self.low = float(low)
410         self.high = float(high)
411         self.nbins = nbins
412         self.binsize=(self.high-self.low)/nbins
413         self._nrObs=0
414         self._sum=0
415         self[:] =[[low+(i-1)*self.binsize,0] for i in range(self.nbins+2)]
416
417     def addIn(self,y):
418         """ add a value into the correct bin"""
419         self._nrObs+=1
420         self._sum+=y
421         b = int ((y-self.low+self.binsize)/self.binsize)
422         if b < 0: b = 0
423         if b > self.nbins+1: b = self.nbins+1
424         assert 0 <= b <=self.nbins+1,'Histogram.addIn: b out of range: %s'%b
425         self[b][1]+=1
426
427     def __str__(self):
428         histo=self
429         ylab="value"
430         nrObs=self._nrObs
431         width=len(str(nrObs))
432         res=[]
433         res.append("<Histogram %s:"%self.name)

```

```

434         res.append("\nNumber of observations: %s\nnrObs)
435     if nrObs:
436         su=self._sum
437         cum=histo[0][1]
438         fmt="%s"
439         line="\n%s <= %s < %s: %s (cum: %s/%s%s)"\
440             %(fmt,"%s",fmt,"%s","%s","%5.1f","%s")
441         line1="\n%s < %s: %s (cum: %s/%s%s)"\
442             %("%s","%s",fmt,"%s","%s","%5.1f","%s")
443         llwidth=len(("< %s <=" % fmt) % histo[1][0])
444         res.append(line1\
445             % (" *llwidth,ylab,histo[1][0],str(histo[0][1]).rjust(width),\
446                 str(cum).rjust(width), (float(cum)/nrObs)*100,"%")
447             )
448         for i in range(1,len(histo)-1):
449             cum+=histo[i][1]
450             res.append(line\
451                 % (histo[i][0],ylab,histo[i+1][0],str(histo[i][1]).rjust(width),\
452                     str(cum).rjust(width), (float(cum)/nrObs)*100,"%")
453                 )
454             cum+=histo[-1][1]
455             linen="\n%s <= %s %s : %s (cum: %s/%s%s)"\
456                 %(fmt,"%s","%s","%s","%s","%5.1f","%s")
457             lnwidth=len(("< %s %s <=" % fmt) % histo[1][0])
458             res.append(linen\
459                 % (histo[-1][0],ylab," *lnwidth,str(histo[-1][1]).rjust(width),\
460                     str(cum).rjust(width), (float(cum)/nrObs)*100,"%")
461                 )
462         res.append("\n>")
463     return " ".join(res)
464
465 def startCollection(when=0.0,monitors=None,tallies=None):
466     """Starts data collection of all designated Monitor and Tally objects
467     (default=all) at time 'when'.
468     """
469     class Starter(Process):
470         def collect(self,monitors,tallies):
471             for m in monitors:
472                 print m.name
473                 m.reset()
474             for t in tallies:
475                 t.reset()
476             yield hold,self
477     if monitors is None:
478         monitors=allMonitors
479     if tallies is None:
480         tallies=allTallies
481     s=Starter()
482     activate(s,s.collect(monitors=monitors,tallies=tallies),at=when)
483
484 class Monitor(list):
485     """ Monitored variables
486
487     A Class for monitored variables, that is, variables that allow one
488     to gather simple statistics. A Monitor is a subclass of list and
489     list operations can be performed on it. An object is established
490     using m= Monitor(name = '..'). It can be given a
491     unique name for use in debugging and in tracing and ylab and tlab
492     strings for labelling graphs.
493     """
494     def __init__(self,name='a_Monitor',ylab='y',tlab='t'):
495         list.__init__(self)
496         self.startTime = 0.0
497         self.name = name
498         self.ylab = ylab
499         self.tlab = tlab
500         allMonitors.append(self)
501
502     def setHistogram(self,name = '',low=0.0,high=100.0,nbins=10):
503         """Sets histogram parameters.
504         Must be called before call to getHistogram"""
505         if name=='':
506             histname=self.name
507         else:
508             histname=name
509         self.histo=Histogram(name=histname,low=low,high=high,nbins=nbins)
510
511     def observe(self,y,t=None):
512         """record y and t"""
513         if t is None: t = now()
514         self.append([t,y])
515
516     def tally(self,y):
517         """ deprecated: tally for backward compatibility"""
518         self.observe(y,0)
519
520     def accum(self,y,t=None):
521         """ deprecated: accum for backward compatibility"""
522         self.observe(y,t)
523
524     def reset(self,t=None):
525         """reset the sums and counts for the monitored variable """

```

```

526         self[:]=[]
527         if t is None: t = now()
528         self.startTime = t
529
530     def tseries(self):
531         """ the series of measured times"""
532         return list(zip(*self)[0])
533
534     def yseries(self):
535         """ the series of measured values"""
536         return list(zip(*self)[1])
537
538     def count(self):
539         """ deprecated: the number of observations made """
540         return self.__len__()
541
542     def total(self):
543         """ the sum of the y"""
544         if self.__len__()==0: return 0
545         else:
546             sum = 0.0
547             for i in range(self.__len__()):
548                 sum += self[i][1]
549             return sum # replace by sum() later
550
551     def mean(self):
552         """ the simple average of the monitored variable"""
553         try: return 1.0*self.total()/self.__len__()
554         except: print 'SimPy: No observations for mean'
555
556     def var(self):
557         """ the sample variance of the monitored variable """
558         n = len(self)
559         tot = self.total()
560         ssq=0.0
561         ##yy = self.yseries()
562         for i in range(self.__len__()):
563             ssq += self[i][1]**2 # replace by sum() eventually
564         try: return (ssq - float(tot*tot)/n)/n
565         except: print 'SimPy: No observations for sample variance'
566
567     def timeAverage(self,t=None):
568         """ the time-weighted average of the monitored variable.
569
570             If t is used it is assumed to be the current time,
571             otherwise t = now()
572
573         """
574         N = self.__len__()
575         if N == 0:
576             print 'SimPy: No observations for timeAverage'
577             return None
578
579         if t is None: t = now()
580         sum = 0.0
581         tlast = self.startTime
582         #print 'DEBUG: timave ',t,tlast
583         ylast = 0.0
584         for i in range(N):
585             ti,yi = self[i]
586             sum += ylast*(ti-tlast)
587             tlast = ti
588             ylast = yi
589         sum += ylast*(t-tlast)
590         T = t - self.startTime
591         if T == 0:
592             print 'SimPy: No elapsed time for timeAverage'
593             return None
594         #print 'DEBUG: timave ',sum,t,T
595         return sum/float(T)
596
597     def timeVariance(self,t=None):
598         """ the time-weighted Variance of the monitored variable.
599
600             If t is used it is assumed to be the current time,
601             otherwise t = now()
602
603         """
604         N = self.__len__()
605         if N == 0:
606             print 'SimPy: No observations for timeVariance'
607             return None
608
609         if t is None: t = now()
610         sm = 0.0
611         ssq = 0.0
612         tlast = self.startTime
613         # print 'DEBUG: 1 twVar ',t,tlast
614         ylast = 0.0
615         for i in range(N):
616             ti,yi = self[i]
617             sm += ylast*(ti-tlast)
618             ssq += ylast*ylast*(ti-tlast)
619             tlast = ti
620             ylast = yi

```

```

618         sm += ylast*(t-tlast)
619         ssq += ylast*ylast*(t-tlast)
620         T = t - self.startTime
621         if T == 0:
622             print 'SimPy: No elapsed time for timeVariance'
623             return None
624         mn = sm/float(T)
625         # print 'DEBUG: 2 twVar ',ssq,t,T
626         return ssq/float(T) - mn*mn
627
628
629     def histogram(self,low=0.0,high=100.0,nbins=10):
630         """ A histogram of the monitored y data values.
631         """
632         h = Histogram(name=self.name,low=low,high=high,nbins=nbins)
633         ys = self.yseries()
634         for y in ys: h.addIn(y)
635         return h
636
637     def getHistogram(self):
638         """Returns a histogram based on the parameters provided in
639         preceding call to setHistogram.
640         """
641         ys = self.yseries()
642         h=self.histo
643         for y in ys: h.addIn(y)
644         return h
645
646     def printHistogram(self,fmt="%s"):
647         """Returns formatted frequency distribution table string from Monitor.
648         Precondition: setHistogram must have been called.
649         fmt==format of bin range values
650         """
651         try:
652             histo=self.getHistogram()
653         except:
654             raise FatalSimerror("histogramTable: call setHistogram first"
655                                 " for Monitor %s"%self.name)
656
657         ylab=self.ylab
658         nrObs=self.count()
659         width=len(str(nrObs))
660         res=[]
661         res.append("\nHistogram for %s:"%histo.name)
662         res.append("\nNumber of observations: %s"%nrObs)
663         su=sum(self.yseries())
664         cum=histo[0][1]
665         line="\n%s <= %s < %s: %s (cum: %s/%s)"%
666             (fmt,"%s",fmt,"%s","%s","%5.1f","%s")
667         line1="\n%s < %s: %s (cum: %s/%s)"%
668             ("%s","%s",fmt,"%s","%s","%5.1f","%s")
669         llwidth=len(("<%s"%fmt)%histo[1][0])
670         res.append(line1%
671             ("% " *llwidth,ylab,histo[1][0],str(histo[0][1]).rjust(width),\
672             str(cum).rjust(width),(float(cum)/nrObs)*100,"%")
673             )
674         for i in range(1,len(histo)-1):
675             cum+=histo[i][1]
676             res.append(line%
677                 (histo[i][0],ylab,histo[i+1][0],str(histo[i][1]).rjust(width),\
678                 str(cum).rjust(width),(float(cum)/nrObs)*100,"%")
679                 )
680             cum+=histo[-1][1]
681             linen="\n%s <= %s %s : %s (cum: %s/%s)"%
682                 (fmt,"%s","%s","%s","%s","%5.1f","%s")
683             lnwidth=len(("<%s"%fmt)%histo[1][0])
684             res.append(linen%
685                 (histo[-1][0],ylab," " *lnwidth,str(histo[-1][1]).rjust(width),\
686                 str(cum).rjust(width),(float(cum)/nrObs)*100,"%")
687                 )
688         return " ".join(res)
689
690     class Tally:
691         def __init__(self, name="a_Tally", ylab="y",tlab="t"):
692             self.name = name
693             self.ylab = ylab
694             self.tlab = tlab
695             self.reset()
696             self.startTime = 0.0
697             self.histo = None
698             self.sum = 0.0
699             self._sum_of_squares = 0
700             self._integral = 0.0 # time-weighted sum
701             self._integral2 = 0.0 # time-weighted sum of squares
702             allTallies.append(self)
703
704     def setHistogram(self, name = '',low=0.0,high=100.0,nbins=10):
705         """Sets histogram parameters.
706         Must be called prior to observations initiate data collection
707         for histogram.
708         """
709         if name=='':
710             hname=self.name

```



```

710         else:
711             hname=name
712             self.histo=Histogram(name=hname,low=low,high=high,nbins=nbins)
713
714     def observe(self, y, t=None):
715         if t is None:
716             t = now()
717         self._integral += (t - self._last_timestamp) * self._last_observation
718         yy = self._last_observation * self._last_observation
719         self._integral2 += (t - self._last_timestamp) * yy
720         self._last_timestamp = t
721         self._last_observation = y
722         self._total += y
723         self._count += 1
724         self._sum += y
725         self._sum_of_squares += y * y
726         if self.histo:
727             self.histo.addIn(y)
728
729     def reset(self, t=None):
730         if t is None:
731             t = now()
732         self.startTime = t
733         self._last_timestamp = t
734         self._last_observation = 0.0
735         self._count = 0
736         self._total = 0.0
737         self._integral = 0.0
738         self._integral2 = 0.0
739         self._sum = 0.0
740         self._sum_of_squares = 0.0
741
742     def count(self):
743         return self._count
744
745     def total(self):
746         return self._total
747
748     def mean(self):
749         return 1.0 * self._total / self._count
750
751     def timeAverage(self,t=None):
752         if t is None:
753             t=now()
754         integ=self._integral+(t - self._last_timestamp) * self._last_observation
755         if (t > self.startTime):
756             return 1.0 * integ/(t - self.startTime)
757         else:
758             print 'SimPy: No elapsed time for timeAverage'
759             return None
760
761     def var(self):
762         return 1.0 * (self._sum_of_squares - (1.0 * (self._sum * self._sum)\
763             / self._count)) / (self._count)
764
765     def timeVariance(self,t=None):
766         """ the time-weighted Variance of the Talled variable.
767
768             If t is used it is assumed to be the current time,
769             otherwise t = now()
770         """
771         if t is None:
772             t=now()
773         twAve = self.timeAverage(t)
774         #print 'Tally timeVariance DEBUG: twave:', twAve
775         last = self._last_observation
776         twinteg2=self._integral2+(t - self._last_timestamp) * last * last
777         #print 'Tally timeVariance DEBUG:tinteg2:', twinteg2
778         if (t > self.startTime):
779             return 1.0 * twinteg2/(t - self.startTime) - twAve*twAve
780         else:
781             print 'SimPy: No elapsed time for timeVariance'
782             return None
783
784
785
786     def __len__(self):
787         return self._count
788
789     def __eq__(self, l):
790         return len(l) == self._count
791
792     def getHistogram(self):
793         return self.histo
794
795     def printHistogram(self,fmt="%s"):
796         """Returns formatted frequency distribution table string from Tally.
797         Precondition: setHistogram must have been called.
798         fmt==format of bin range values
799         """
800         try:
801             histo=self.getHistogram()

```

```

802         except:
803             raise FatalSimerror("histogramTable: call setHistogram first"\
804                                 " for Tally %s"%self.name)
805
806         ylab=self.ylab
807         nrObs=self.count()
808         width=len(str(nrObs))
809         res=[]
810         res.append("\nHistogram for %s:"%histo.name)
811         res.append("\nNumber of observations: %s"%nrObs)
812         su=self.total()
813         cum=histo[0][1]
814         line="\n%s <= %s < %s: %s (cum: %s/%s)"% \
815             %(fmt,"%s",fmt,"%s","%s","%5.1f","%s")
816         line1="\n%s < %s: %s (cum: %s/%s)"% \
817             %("%s","%s",fmt,"%s","%s","%5.1f","%s")
818         llwidth=len((" %s <= "%fmt)%histo[1][0])
819         res.append(line1\
820                 %(" " *llwidth,ylab,histo[1][0],str(histo[0][1]).rjust(width),\
821                   str(cum).rjust(width),(float(cum)/nrObs)*100,"%")
822                )
823         for i in range(1,len(histo)-1):
824             cum+=histo[i][1]
825             res.append(line\
826                     % (histo[i][0],ylab,histo[i+1][0],str(histo[i][1]).rjust(width),\
827                       str(cum).rjust(width),(float(cum)/nrObs)*100,"%")
828                    )
829             cum+=histo[-1][1]
830             linen="\n%s <= %s %s : %s (cum: %s/%s)"% \
831                 %(fmt,"%s","%s","%s","%s","%5.1f","%s")
832             lnwidth=len(("<%s"%fmt)%histo[1][0])
833             res.append(linen\
834                     % (histo[-1][0],ylab," " *lnwidth,str(histo[-1][1]).rjust(width),\
835                       str(cum).rjust(width),(float(cum)/nrObs)*100,"%")
836                    )
837         return " ".join(res)
838
839 class Queue(list):
840     def __init__(self,res,moni):
841         if not moni is None: #moni==[:
842             self.monit=True # True if a type of Monitor/Tally attached
843         else:
844             self.monit=False
845         self.monit=moni # The Monitor/Tally
846         self.resource=res # the resource/buffer this queue belongs to
847
848     def enter(self,obj):
849         pass
850
851     def leave(self):
852         pass
853
854     def takeout(self,obj):
855         self.remove(obj)
856         if self.monit:
857             self.monit.observe(len(self),t=now())
858
859 class FIFO(Queue):
860     def __init__(self,res,moni):
861         Queue.__init__(self,res,moni)
862
863     def enter(self,obj):
864         self.append(obj)
865         if self.monit:
866             self.monit.observe(len(self),t=now())
867
868     def enterGet(self,obj):
869         self.enter(obj)
870
871     def enterPut(self,obj):
872         self.enter(obj)
873
874     def leave(self):
875         a= self.pop(0)
876         if self.monit:
877             self.monit.observe(len(self),t=now())
878         return a
879
880 class PriorityQ(FIFO):
881     """Queue is always ordered according to priority.
882     Higher value of priority attribute == higher priority.
883     """
884     def __init__(self,res,moni):
885         FIFO.__init__(self,res,moni)
886
887     def enter(self,obj):
888         """Handles request queue for Resource"""
889         if len(self):
890             ix=self.resource
891             if self[-1].__priority[ix] >= obj.__priority[ix]:
892                 self.append(obj)
893             else:
894                 z=0

```

```

894         while self[z]._priority[ix] >= obj._priority[ix]:
895             z += 1
896         self.insert(z,obj)
897     else:
898         self.append(obj)
899     if self.monit:
900         self.monit.observe(len(self),t=now())
901
902     def enterGet(self,obj):
903         """Handles getQ in Buffer"""
904         if len(self):
905             ix=self.resource
906             #print "priority:",[x._priority[ix] for x in self]
907             if self[-1]._getpriority[ix] >= obj._getpriority[ix]:
908                 self.append(obj)
909             else:
910                 z=0
911                 while self[z]._getpriority[ix] >= obj._getpriority[ix]:
912                     z += 1
913                 self.insert(z,obj)
914             else:
915                 self.append(obj)
916         if self.monit:
917             self.monit.observe(len(self),t=now())
918
919     def enterPut(self,obj):
920         """Handles putQ in Buffer"""
921         if len(self):
922             ix=self.resource
923             #print "priority:",[x._priority[ix] for x in self]
924             if self[-1]._putpriority[ix] >= obj._putpriority[ix]:
925                 self.append(obj)
926             else:
927                 z=0
928                 while self[z]._putpriority[ix] >= obj._putpriority[ix]:
929                     z += 1
930                 self.insert(z,obj)
931             else:
932                 self.append(obj)
933         if self.monit:
934             self.monit.observe(len(self),t=now())
935
936     class Resource(Lister):
937         """Models shared, limited capacity resources with queuing;
938         FIFO is default queuing discipline.
939         """
940
941         def __init__(self,capacity=1,name="a_resource",unitName="units",
942                     qType=FIFO,preemptable=0,monitored=False,monitorType=Monitor):
943             """
944             monitorType=(Monitor(default)|Tally)
945             """
946             self.name=name           # resource name
947             self.capacity=capacity    # resource units in this resource
948             self.unitName=unitName    # type name of resource units
949             self.n=capacity           # uncommitted resource units
950             self.monitored=monitored
951
952             if self.monitored:        # Monitor waitQ, activeQ
953                 self.actMon=monitorType(name="Active Queue Monitor %s"%self.name,
954                                         ylab="nr in queue",tlab="time")
955                 monact=self.actMon
956                 self.waitMon=monitorType(name="Wait Queue Monitor %s"%self.name,
957                                         ylab="nr in queue",tlab="time")
958                 monwait=self.waitMon
959             else:
960                 monwait=None
961                 monact=None
962             self.waitQ=qType(self,monwait)
963             self.preemptable=preemptable
964             self.activeQ=qType(self,monact)
965             self.priority_default=0
966
967         def _request(self,arg):
968             """Process request event for this resource"""
969             obj=arg[1]
970             if len(arg[0]) == 4:          # yield request,self,resource,priority
971                 obj._priority[self]=arg[0][3]
972             else:                          # yield request,self,resource
973                 obj._priority[self]=self.priority_default
974             if self.preemptable and self.n == 0: # No free resource
975                 # test for preemption condition
976                 preempt=obj._priority[self] > self.activeQ[-1]._priority[self]
977                 # If yes:
978                 if preempt:
979                     z=self.activeQ[-1]
980                     # suspend lowest priority process being served
981                     ##suspended = z
982                     # record remaining service time
983                     z._remainService = z._nextTime - _t
984                     Process().cancel(z)
985                     # remove from activeQ

```

```

986         self.activeQ.remove(z)
987         # put into front of waitQ
988         self.waitQ.insert(0,z)
989         # if self is monitored, update waitQ monitor
990         if self.monitored:
991             self.waitMon.observe(len(self.waitQ),now())
992         # record that it has been preempted
993         z._preempted = 1
994         # passivate re-queued process
995         z._nextTime=None
996         # assign resource unit to preemptor
997         self.activeQ.enter(obj)
998         # post event notice for preempting process
999         _e._post(obj,at=_t,prior=1)
1000     else:
1001         self.waitQ.enter(obj)
1002         # passivate queuing process
1003         obj._nextTime=None
1004     else: # treat non-preemption case
1005         if self.n == 0:
1006             self.waitQ.enter(obj)
1007             # passivate queuing process
1008             obj._nextTime=None
1009         else:
1010             self.n -= 1
1011             self.activeQ.enter(obj)
1012             _e._post(obj,at=_t,prior=1)
1013
1014     def _release(self,arg):
1015         """Process release request for this resource"""
1016         self.n += 1
1017         self.activeQ.remove(arg[1])
1018         if self.monitored:
1019             self.actMon.observe(len(self.activeQ),t=now())
1020         #reactivate first waiting requestor if any; assign Resource to it
1021         if self.waitQ:
1022             obj=self.waitQ.leave()
1023             self.n -= 1 #assign 1 resource unit to object
1024             self.activeQ.enter(obj)
1025             # if resource preemptable:
1026             if self.preemptable:
1027                 # if object had been preempted:
1028                 if obj._preempted:
1029                     obj._preempted = 0
1030                     # reactivate object delay= remaining service time
1031                     reactivate(obj,delay=obj._remainService)
1032                 # else reactivate right away
1033                 else:
1034                     reactivate(obj,delay=0,prior=1)
1035             # else:
1036             else:
1037                 reactivate(obj,delay=0,prior=1)
1038             _e._post(arg[1],at=_t,prior=1)
1039
1040     class Buffer(Lister):
1041         """Abstract class for buffers
1042         Blocks a process when a put would cause buffer overflow or a get would cause
1043         buffer underflow.
1044         Default queuing discipline for blocked processes is FIFO."""
1045
1046         priorityDefault=0
1047         def __init__(self,name=None,capacity="unbounded",unitName="units",
1048                     putQType=FIFO,getQType=FIFO,
1049                     monitored=False,monitorType=Monitor,initialBuffered=None):
1050             if capacity=="unbounded": capacity=sys.maxint
1051             self.capacity=capacity
1052             self.name=name
1053             self.putQType=putQType
1054             self.getQType=getQType
1055             self.monitored=monitored
1056             self.initialBuffered=initialBuffered
1057             self.unitName=unitName
1058             if self.monitored:
1059                 ## monitor for Producer processes' queue
1060                 self.putQMon=monitorType(name="Producer Queue Monitor %s"%self.name,
1061                                         ylab="nr in queue",tlab="time")
1062                 ## monitor for Consumer processes' queue
1063                 self.getQMon=monitorType(name="Consumer Queue Monitor %s"%self.name,
1064                                         ylab="nr in queue",tlab="time")
1065                 ## monitor for nr items in buffer
1066                 self.bufferMon=monitorType(name="Buffer Monitor %s"%self.name,
1067                                           ylab="nr in buffer",tlab="time")
1068             else:
1069                 self.putQMon=None
1070                 self.getQMon=None
1071                 self.bufferMon=None
1072             self.putQ=self.putQType(res=self,moni=self.putQMon)
1073             self.getQ=self.getQType(res=self,moni=self.getQMon)
1074             if self.monitored:
1075                 self.putQMon.observe(y=len(self.putQ),t=now())
1076                 self.getQMon.observe(y=len(self.getQ),t=now())
1077             self._putpriority={}

```

```

1078     self._getpriority={}
1079
1080     def _put(self):
1081         pass
1082     def _get(self):
1083         pass
1084
1085 class Level(Buffer):
1086     """Models buffers for processes putting/getting un-distinguishable items.
1087     """
1088     def getamount(self):
1089         return self.nrBuffered
1090
1091     def gettheBuffer(self):
1092         return self.nrBuffered
1093
1094     theBuffer=property(gettheBuffer)
1095
1096     def __init__(self,**pars):
1097         Buffer.__init__(self,**pars)
1098         if self.name is None:
1099             self.name="a_level"    ## default name
1100
1101         if (type(self.capacity)!=type(1.0) and\
1102             type(self.capacity)!=type(1)) or\
1103             self.capacity<0:
1104             raise FatalSimerror\
1105                 ("Level: capacity parameter not a positive number: %s"\
1106                  %self.initialBuffered)
1107
1108         if type(self.initialBuffered)==type(1.0) or\
1109             type(self.initialBuffered)==type(1):
1110             if self.initialBuffered>self.capacity:
1111                 raise FatalSimerror("initialBuffered exceeds capacity")
1112             if self.initialBuffered>=0:
1113                 self.nrBuffered=self.initialBuffered ## nr items initially in buffer
1114                 ## buffer is just a counter (int type)
1115             else:
1116                 raise FatalSimerror\
1117                     ("initialBuffered param of Level negative: %s"\
1118                      %self.initialBuffered)
1119         elif self.initialBuffered is None:
1120             self.initialBuffered=0
1121             self.nrBuffered=0
1122         else:
1123             raise FatalSimerror\
1124                 ("Level: wrong type of initialBuffered (parameter=%s)"\
1125                  %self.initialBuffered)
1126         if self.monitored:
1127             self.bufferMon.observe(y=self.amount,t=now())
1128     amount=property(getamount)
1129
1130     def _put(self,arg):
1131         """Handles put requests for Level instances"""
1132         obj=arg[1]
1133         if len(arg[0]) == 5:      # yield put,self,buff,whattoput,priority
1134             obj._putpriority[self]=arg[0][4]
1135             whatToPut=arg[0][3]
1136         elif len(arg[0]) == 4:   # yield get,self,buff,whattoput
1137             obj._putpriority[self]=Buffer.priorityDefault #default
1138             whatToPut=arg[0][3]
1139         else:                    # yield get,self,buff
1140             obj._putpriority[self]=Buffer.priorityDefault #default
1141             whatToPut=1
1142         if type(whatToPut)!=type(1) and type(whatToPut)!=type(1.0):
1143             raise FatalSimerror("Level: put parameter not a number")
1144         if not whatToPut>=0.0:
1145             raise FatalSimerror("Level: put parameter not positive number")
1146         whatToPutNr=whatToPut
1147         if whatToPutNr+self.amount>self.capacity:
1148             obj._nextTime=None      #passivate put requestor
1149             obj._whatToPut=whatToPutNr
1150             self.putQ.enterPut(obj)  #and queue, with size of put
1151         else:
1152             self.nrBuffered+=whatToPutNr
1153             if self.monitored:
1154                 self.bufferMon.observe(y=self.amount,t=now())
1155             # service any getters waiting
1156             # service in queue-order; do not serve second in queue before first
1157             # has been served
1158             while len(self.getQ) and self.amount>0:
1159                 proc=self.getQ[0]
1160                 if proc._nrToGet<=self.amount:
1161                     proc.got=proc._nrToGet
1162                     self.nrBuffered-=proc.got
1163                     if self.monitored:
1164                         self.bufferMon.observe(y=self.amount,t=now())
1165                     self.getQ.takeout(proc) # get requestor's record out of queue
1166                     _e._post(proc,at=_t) # continue a blocked get requestor
1167                 else:
1168                     break
1169             _e._post(obj,at=_t,prior=1) # continue the put requestor

```

```

1170
1171 def _get(self, arg):
1172     """Handles get requests for Level instances"""
1173     obj=arg[1]
1174     obj.got=None
1175     if len(arg[0]) == 5:         # yield get, self, buff, whattoget, priority
1176         obj._getpriority[self]=arg[0][4]
1177         nrToGet=arg[0][3]
1178     elif len(arg[0]) == 4:     # yield get, self, buff, whattoget
1179         obj._getpriority[self]=Buffer.priorityDefault #default
1180         nrToGet=arg[0][3]
1181     else:                       # yield get, self, buff
1182         obj._getpriority[self]=Buffer.priorityDefault
1183         nrToGet=1
1184     if type(nrToGet)!=type(1.0) and type(nrToGet)!=type(1):
1185         raise FatalSimerror\
1186             ("Level: get parameter not a number: %s"%nrToGet)
1187     if nrToGet<0:
1188         raise FatalSimerror\
1189             ("Level: get parameter not positive number: %s"%nrToGet)
1190     if self.amount < nrToGet:
1191         obj._nrToGet=nrToGet
1192         self.getQ.enterGet(obj)
1193         # passivate queuing process
1194         obj._nextTime=None
1195     else:
1196         obj.got=nrToGet
1197         self.nrBuffered-=nrToGet
1198         if self.monitored:
1199             self.bufferMon.observe(y=self.amount, t=now())
1200             _e._post(obj, at=_t, prior=1)
1201         # reactivate any put requestors for which space is now available
1202         # service in queue-order; do not serve second in queue before first
1203         # has been served
1204         while len(self.putQ): #test for queued producers
1205             proc=self.putQ[0]
1206             if proc._whatToPut+self.amount<=self.capacity:
1207                 self.nrBuffered+=proc._whatToPut
1208                 if self.monitored:
1209                     self.bufferMon.observe(y=self.amount, t=now())
1210                 self.putQ.takeout(proc)#requestor's record out of queue
1211                 _e._post(proc, at=_t) # continue a blocked put requestor
1212             else:
1213                 break
1214
1215 class Store(Buffer):
1216     """Models buffers for processes coupled by putting/getting distinguishable
1217     items.
1218     Blocks a process when a put would cause buffer overflow or a get would cause
1219     buffer underflow.
1220     Default queuing discipline for blocked processes is priority FIFO.
1221     """
1222     def getnrBuffered(self):
1223         return len(self.theBuffer)
1224     nrBuffered=property(getnrBuffered)
1225
1226     def getbuffered(self):
1227         return self.theBuffer
1228     buffered=property(getbuffered)
1229
1230     def __init__(self, **pars):
1231         Buffer.__init__(self, **pars)
1232         self.theBuffer=[]
1233         if self.name is None:
1234             self.name="a_store" ## default name
1235         if type(self.capacity)!=type(1) or self.capacity<=0:
1236             raise FatalSimerror\
1237                 ("Store: capacity parameter not a positive integer > 0: %s"%
1238                  self.initialBuffered)
1239         if type(self.initialBuffered)==type([]):
1240             if len(self.initialBuffered)>self.capacity:
1241                 raise FatalSimerror("initialBuffered exceeds capacity")
1242             else:
1243                 self.theBuffer[:]=self.initialBuffered##buffer==list of objects
1244         elif self.initialBuffered is None:
1245             self.theBuffer=[]
1246         else:
1247             raise FatalSimerror\
1248                 ("Store: initialBuffered not a list")
1249         if self.monitored:
1250             self.bufferMon.observe(y=self.nrBuffered, t=now())
1251         self._sort=None
1252
1253
1254
1255     def addSort(self, sortFunc):
1256         """Adds buffer sorting to this instance of Store. It maintains
1257         theBuffer sorted by the sortAttr attribute of the objects in the
1258         buffer.
1259         The user-provided 'sortFunc' must look like this:
1260
1261         def mySort(self, par):

```

```

1262         tmplist=[(x.sortAttr,x) for x in par]
1263         tmplist.sort()
1264         return [x for (key,x) in tmplist]
1265
1266     """
1267
1268     self._sort=new.instancemethod(sortFunc,self,self.__class__)
1269     self.theBuffer=self._sort(self.theBuffer)
1270
1271 def _put(self,arg):
1272     """Handles put requests for Store instances"""
1273     obj=arg[1]
1274     if len(arg[0]) == 5:         # yield put,self,buff,whattoput,priority
1275         obj._putpriority[self]=arg[0][4]
1276         whatToPut=arg[0][3]
1277     elif len(arg[0]) == 4:     # yield put,self,buff,whattoput
1278         obj._putpriority[self]=Buffer.priorityDefault #default
1279         whatToPut=arg[0][3]
1280     else:                       # error, whattopt missing
1281         raise FatalSimerror("Item to put missing in yield put stmt")
1282     if type(whatToPut)!=type([]):
1283         raise FatalSimerror("put parameter is not a list")
1284     whatToPutNr=len(whatToPut)
1285     if whatToPutNr+self.nrBuffered>self.capacity:
1286         obj._nextTime=None     #passivate put requestor
1287         obj._whatToPut=whatToPut
1288         self.putQ.enterPut(obj) #and queue, with items to put
1289     else:
1290         self.theBuffer.extend(whatToPut)
1291         if not (self._sort is None):
1292             self.theBuffer=self._sort(self.theBuffer)
1293         if self.monitored:
1294             self.bufferMon.observe(y=self.nrBuffered,t=now())
1295
1296     # service any waiting getters
1297     # service in queue order: do not serve second in queue before first
1298     # has been served
1299     while self.nrBuffered>0 and len(self.getQ):
1300         proc=self.getQ[0]
1301         if inspect.isfunction(proc._nrToGet):
1302             movCand=proc._nrToGet(self.theBuffer) #predicate parameter
1303             if movCand:
1304                 proc.got=movCand[:]
1305                 for i in movCand:
1306                     self.theBuffer.remove(i)
1307                 self.getQ.takeout(proc)
1308                 if self.monitored:
1309                     self.bufferMon.observe(y=self.nrBuffered,t=now())
1310                 _e._post(what=proc,at=_t) # continue a blocked get requestor
1311             else:
1312                 break
1313         else: #numerical parameter
1314             if proc._nrToGet<=self.nrBuffered:
1315                 nrToGet=proc._nrToGet
1316                 proc.got=[]
1317                 proc.got[:]=self.theBuffer[0:nrToGet]
1318                 self.theBuffer[:]=self.theBuffer[nrToGet:]
1319                 if self.monitored:
1320                     self.bufferMon.observe(y=self.nrBuffered,t=now())
1321                 # take this get requestor's record out of queue:
1322                 self.getQ.takeout(proc)
1323                 _e._post(what=proc,at=_t) # continue a blocked get requestor
1324             else:
1325                 break
1326
1327         _e._post(what=obj,at=_t,prior=1) # continue the put requestor
1328
1329 def _get(self,arg):
1330     """Handles get requests"""
1331     filtfunc=None
1332     obj=arg[1]
1333     obj.got=[]
1334     if len(arg[0]) == 5:         # the list of items retrieved by 'get'
1335         # yield get,self,buff,whattoget,priority
1336         obj._getpriority[self]=arg[0][4]
1337         if inspect.isfunction(arg[0][3]):
1338             filtfunc=arg[0][3]
1339         else:
1340             nrToGet=arg[0][3]
1341     elif len(arg[0]) == 4:     # yield get,self,buff,whattoget
1342         obj._getpriority[self]=Buffer.priorityDefault #default
1343         if inspect.isfunction(arg[0][3]):
1344             filtfunc=arg[0][3]
1345         else:
1346             nrToGet=arg[0][3]
1347     else:                       # yield get,self,buff
1348         obj._getpriority[self]=Buffer.priorityDefault
1349         nrToGet=1
1350     if not filtfunc: #number specifies nr items to get
1351         if nrToGet<0:
1352             raise FatalSimerror\
1353                 ("Store: get parameter not positive number: %s"%nrToGet)
1354     if self.nrBuffered < nrToGet:

```

```

1354         obj._nrToGet=nrToGet
1355         self.getQ.enterGet(obj)
1356         # passivate/block queuing 'get' process
1357         obj._nextTime=None
1358     else:
1359         for i in range(nrToGet):
1360             obj.got.append(self.theBuffer.pop(0)) # move items from
1361                                                     # buffer to requesting process
1362         if self.monitored:
1363             self.bufferMon.observe(y=self.nrBuffered,t=now())
1364         _e._post(obj,at=_t,prior=1)
1365         # reactivate any put requestors for which space is now available
1366         # serve in queue order: do not serve second in queue before first
1367         # has been served
1368         while len(self.putQ):
1369             proc=self.putQ[0]
1370             if len(proc._whatToPut)+self.nrBuffered<=self.capacity:
1371                 for i in proc._whatToPut:
1372                     self.theBuffer.append(i) #move items to buffer
1373                 if not(self._sort is None):
1374                     self.theBuffer=self._sort(self.theBuffer)
1375                 if self.monitored:
1376                     self.bufferMon.observe(y=self.nrBuffered,t=now())
1377                 self.putQ.takeout(proc) # dequeue requestor's record
1378                 _e._post(proc,at=_t) # continue a blocked put requestor
1379             else:
1380                 break
1381     else: # items to get determined by filtfunc
1382         movCand=filtfunc(self.theBuffer)
1383         if movCand: # get succeeded
1384             _e._post(obj,at=_t,prior=1)
1385             obj.got=movCand[:]
1386             for item in movCand:
1387                 self.theBuffer.remove(item)
1388             if self.monitored:
1389                 self.bufferMon.observe(y=self.nrBuffered,t=now())
1390             # reactivate any put requestors for which space is now available
1391             # serve in queue order: do not serve second in queue before first
1392             # has been served
1393             while len(self.putQ):
1394                 proc=self.putQ[0]
1395                 if len(proc._whatToPut)+self.nrBuffered<=self.capacity:
1396                     for i in proc._whatToPut:
1397                         self.theBuffer.append(i) #move items to buffer
1398                     if not(self._sort is None):
1399                         self.theBuffer=self._sort(self.theBuffer)
1400                     if self.monitored:
1401                         self.bufferMon.observe(y=self.nrBuffered,t=now())
1402                     self.putQ.takeout(proc) # dequeue requestor's record
1403                     _e._post(proc,at=_t) # continue a blocked put requestor
1404                 else:
1405                     break
1406             else: # get did not succeed, block
1407                 obj._nrToGet=filtfunc
1408                 self.getQ.enterGet(obj)
1409                 # passivate/block queuing 'get' process
1410                 obj._nextTime=None
1411
1412 class SimEvent(Lister):
1413     """Supports one-shot signalling between processes. All processes waiting for an event to occur
1414     get activated when its occurrence is signalled. From the processes queuing for an event, only
1415     the first gets activated.
1416     """
1417     def __init__(self,name="a_SimEvent"):
1418         self.name=name
1419         self.waits=[]
1420         self.queues=[]
1421         self.occurred=False
1422         self.signalparam=None
1423
1424     def signal(self,param=None):
1425         """Produces a signal to self;
1426         Fires this event (makes it occur).
1427         Reactivates ALL processes waiting for this event. (Cleanup waits lists
1428         of other events if wait was for an event-group (OR).)
1429         Reactivates the first process for which event(s) it is queuing for
1430         have fired. (Cleanup queues of other events if wait was for an event-group (OR).)
1431         """
1432         self.signalparam=param
1433         if not self.waits and not self.queues:
1434             self.occurred=True
1435         else:
1436             #reactivate all waiting processes
1437             for p in self.waits:
1438                 p[0].eventsFired.append(self)
1439                 reactivate(p[0],prior=True)
1440             #delete waits entries for this process in other events
1441             for ev in p[1]:
1442                 if ev!=self:
1443                     if ev.occurred:
1444                         p[0].eventsFired.append(ev)
1445             for iev in ev.waits:

```



```

1446             if iev[0]==p[0]:
1447                 ev.waits.remove(iev)
1448                 break
1449         self.waits=[]
1450         if self.queues:
1451             proc=self.queues.pop(0)[0]
1452             proc.eventsFired.append(self)
1453             reactivate(proc)
1454
1455     def _wait(self,par):
1456         """Consumes a signal if it has occurred, otherwise process 'proc'
1457         waits for this event.
1458         """
1459         proc=par[0][1] #the process issuing the yield waitevent command
1460         proc.eventsFired=[]
1461         if not self.occurred:
1462             self.waits.append([proc,[self]])
1463             proc._nextTime=None #passivate calling process
1464         else:
1465             proc.eventsFired.append(self)
1466             self.occurred=False
1467             _e._post(proc,at=_t,prior=1)
1468
1469     def _waitOR(self,par):
1470         """Handles waiting for an OR of events in a tuple/list.
1471         """
1472         proc=par[0][1]
1473         evlist=par[0][2]
1474         proc.eventsFired=[]
1475         anyoccur=False
1476         for ev in evlist:
1477             if ev.occurred:
1478                 anyoccur=True
1479                 proc.eventsFired.append(ev)
1480                 ev.occurred=False
1481         if anyoccur: #at least one event has fired; continue process
1482             _e._post(proc,at=_t,prior=1)
1483
1484         else: #no event in list has fired, enter process in all 'waits' lists
1485             proc.eventsFired=[]
1486             proc._nextTime=None #passivate calling process
1487             for ev in evlist:
1488                 ev.waits.append([proc,evlist])
1489
1490     def _queue(self,par):
1491         """Consumes a signal if it has occurred, otherwise process 'proc'
1492         queues for this event.
1493         """
1494         proc=par[0][1] #the process issuing the yield queueevent command
1495         proc.eventsFired=[]
1496         if not self.occurred:
1497             self.queues.append([proc,[self]])
1498             proc._nextTime=None #passivate calling process
1499         else:
1500             proc.eventsFired.append(self)
1501             self.occurred=False
1502             _e._post(proc,at=_t,prior=1)
1503
1504     def _queueOR(self,par):
1505         """Handles queueing for an OR of events in a tuple/list.
1506         """
1507         proc=par[0][1]
1508         evlist=par[0][2]
1509         proc.eventsFired=[]
1510         anyoccur=False
1511         for ev in evlist:
1512             if ev.occurred:
1513                 anyoccur=True
1514                 proc.eventsFired.append(ev)
1515                 ev.occurred=False
1516         if anyoccur: #at least one event has fired; continue process
1517             _e._post(proc,at=_t,prior=1)
1518
1519         else: #no event in list has fired, enter process in all 'waits' lists
1520             proc.eventsFired=[]
1521             proc._nextTime=None #passivate calling process
1522             for ev in evlist:
1523                 ev.queues.append([proc,evlist])
1524
1525     ## begin waituntil functionality
1526     def _test():
1527         """
1528         Gets called by simulate after every event, as long as there are processes
1529         waiting in condQ for a condition to be satisfied.
1530         Tests the conditions for all waiting processes. Where condition satisfied,
1531         reactivates that process immediately and removes it from queue.
1532         """
1533         global condQ
1534         rList=[]
1535         for el in condQ:
1536             if el.cond():
1537                 rList.append(el)

```

```

1538         reactivate(e1)
1539     for i in rList:
1540         condQ.remove(i)
1541
1542     if not condQ:
1543         _stopWUStepping()
1544
1545 def _waitUntilFunc(proc,cond):
1546     global condQ
1547     """
1548     Puts a process 'proc' waiting for a condition into a waiting queue.
1549     'cond' is a predicate function which returns True if the condition is
1550     satisfied.
1551     """
1552     if not cond():
1553         condQ.append(proc)
1554         proc.cond=cond
1555         _startWUStepping()           #signal 'simulate' that a process is waiting
1556         # passivate calling process
1557         proc._nextTime=None
1558     else:
1559         #schedule continuation of calling process
1560         _e._post(proc,at=_t,prior=1)
1561
1562
1563 ##end waituntil functionality
1564
1565 def scheduler(till=0):
1566     """Schedules Processes/semi-coroutines until time 'till'.
1567     Deprecated since version 0.5.
1568     """
1569     simulate(until=till)
1570
1571 def holdfunc(a):
1572     a[0][1]._hold(a)
1573
1574 def requestfunc(a):
1575     """Handles 'yield request,self,res' and 'yield (request,self,res),(<code>,self,par)'.
1576     <code> can be 'hold' or 'waitevent'.
1577     """
1578     if type(a[0][0])==tuple:
1579         ## Compound yield request statement
1580         ## first tuple in ((request,self,res),(xx,self,yy))
1581         b=a[0][0]
1582         ## b[2]==res (the resource requested)
1583         ##process the first part of the compound yield statement
1584         ##a[1] is the Process instance
1585         b[2]._request(arg=(b,a[1]))
1586         ##deal with add-on condition to command
1587         ##Trigger processes for renegeing
1588         class _Holder(Process):
1589             """Provides timeout process"""
1590             def trigger(self,delay):
1591                 yield hold,self,delay
1592                 if not proc in b[2].activeQ:
1593                     reactivate(proc)
1594
1595         class _EventWait(Process):
1596             """Provides event waiting process"""
1597             def trigger(self,event):
1598                 yield waitevent,self,event
1599                 if not proc in b[2].activeQ:
1600                     a[1].eventsFired=self.eventsFired
1601                     reactivate(proc)
1602
1603         #activate it
1604         proc=a[0][0][1] # the process to be woken up
1605         actCode=a[0][1][0]
1606         if actCode==hold:
1607             proc._holder=_Holder(name="RENEGE-hold for %s"%proc.name)
1608             ## the timeout delay
1609             activate(proc._holder,proc._holder.trigger(a[0][1][2]))
1610         elif actCode==waituntil:
1611             raise FatalSimerror("Illegal code for renegeing: waituntil")
1612         elif actCode==waitevent:
1613             proc._holder=_EventWait(name="RENEGE-waitevent for %s"%proc.name)
1614             ## the event
1615             activate(proc._holder,proc._holder.trigger(a[0][1][2]))
1616         elif actCode==queueevent:
1617             raise FatalSimerror("Illegal code for renegeing: queueevent")
1618         else:
1619             raise FatalSimerror("Illegal code for renegeing %s"%actCode)
1620     else:
1621         ## Simple yield request command
1622         a[0][2]._request(a)
1623
1624 def releasefunc(a):
1625     a[0][2]._release(a)
1626
1627 def passivatefunc(a):
1628     a[0][1]._passivate(a)
1629

```

```

1630 def waitevfunc(a):
1631     #if waiting for one event only (not a tuple or list)
1632     evtpar=a[0][2]
1633     if isinstance(evtpar,SimEvent):
1634         a[0][2]._wait(a)
1635     # else, if waiting for an OR of events (list/tuple):
1636     else: #it should be a list/tuple of events
1637         # call _waitOR for first event
1638         evtpar[0]._waitOR(a)
1639
1640 def queueevfunc(a):
1641     #if queueing for one event only (not a tuple or list)
1642     evtpar=a[0][2]
1643     if isinstance(evtpar,SimEvent):
1644         a[0][2]._queue(a)
1645     #else, if queueing for an OR of events (list/tuple):
1646     else: #it should be a list/tuple of events
1647         # call _queueOR for first event
1648         evtpar[0]._queueOR(a)
1649
1650 def waituntilfunc(par):
1651     _waitUntilFunc(par[0][1],par[0][2])
1652
1653 def getfunc(a):
1654     """Handles 'yield get,self,buffer,what,priority' and
1655     'yield (get,self,buffer,what,priority),(<code>,self,par)'.
1656     <code> can be 'hold' or 'waitevent'.
1657     """
1658     if type(a[0][0])==tuple:
1659         ## Compound yield request statement
1660         ## first tuple in ((request,self,res),(xx,self,yy))
1661         b=a[0][0]
1662         ## b[2]==res (the resource requested)
1663         ##process the first part of the compound yield statement
1664         ##a[1] is the Process instance
1665         b[2]._get(arg=(b,a[1]))
1666         ##deal with add-on condition to command
1667         ##Trigger processes for renegeing
1668         class _Holder(Process):
1669             """Provides timeout process"""
1670             def trigger(self,delay):
1671                 yield hold,self,delay
1672                 #if not proc in b[2].activeQ:
1673                 if proc in b[2].getQ:
1674                     reactivate(proc)
1675
1676         class _EventWait(Process):
1677             """Provides event waiting process"""
1678             def trigger(self,event):
1679                 yield waitevent,self,event
1680                 if proc in b[2].getQ:
1681                     a[1].eventsFired=self.eventsFired
1682                     reactivate(proc)
1683
1684         #activate it
1685         proc=a[0][0][1] # the process to be woken up
1686         actCode=a[0][1][0]
1687         if actCode==hold:
1688             proc._holder=_Holder("RENEGE-hold for %s"%proc.name)
1689             ## the timeout delay
1690             activate(proc._holder,proc._holder.trigger(a[0][1][2]))
1691         elif actCode==waituntil:
1692             raise FatalSimerror("Illegal code for renegeing: waituntil")
1693         elif actCode==waitevent:
1694             proc._holder=_EventWait(proc.name)
1695             ## the event
1696             activate(proc._holder,proc._holder.trigger(a[0][1][2]))
1697         elif actCode==queueevent:
1698             raise FatalSimerror("Illegal code for renegeing: queueevent")
1699         else:
1700             raise FatalSimerror("Illegal code for renegeing %s"%actCode)
1701     else:
1702         ## Simple yield request command
1703         a[0][2]._get(a)
1704
1705
1706 def putfunc(a):
1707     """Handles 'yield put' (simple and compound hold/waitevent)
1708     """
1709     if type(a[0][0])==tuple:
1710         ## Compound yield request statement
1711         ## first tuple in ((request,self,res),(xx,self,yy))
1712         b=a[0][0]
1713         ## b[2]==res (the resource requested)
1714         ##process the first part of the compound yield statement
1715         ##a[1] is the Process instance
1716         b[2]._put(arg=(b,a[1]))
1717         ##deal with add-on condition to command
1718         ##Trigger processes for renegeing
1719         class _Holder(Process):
1720             """Provides timeout process"""
1721             def trigger(self,delay):

```

```

1722         yield hold,self,delay
1723         #if not proc in b[2].activeQ:
1724         if proc in b[2].putQ:
1725             reactivate(proc)
1726
1727     class _EventWait(Process):
1728         """Provides event waiting process"""
1729         def trigger(self,event):
1730             yield waitevent,self,event
1731             if proc in b[2].putQ:
1732                 a[1].eventsFired=self.eventsFired
1733                 reactivate(proc)
1734
1735         #activate it
1736         proc=a[0][0][1] # the process to be woken up
1737         actCode=a[0][1][0]
1738         if actCode==hold:
1739             proc._holder=_Holder("RENEGE-hold for %s"%proc.name)
1740             ## the timeout delay
1741             activate(proc._holder,proc._holder.trigger(a[0][1][2]))
1742         elif actCode==waituntil:
1743             raise FatalSimerror("Illegal code for renegeing: waituntil")
1744         elif actCode==waitevent:
1745             proc._holder=_EventWait("RENEGE-waitevent for %s"%proc.name)
1746             ## the event
1747             activate(proc._holder,proc._holder.trigger(a[0][1][2]))
1748         elif actCode==queueevent:
1749             raise FatalSimerror("Illegal code for renegeing: queueevent")
1750         else:
1751             raise FatalSimerror("Illegal code for renegeing %s"%actCode)
1752     else:
1753         ## Simple yield request command
1754         a[0][2]._put(a)
1755
1756 def simulate(until=0):
1757     """Schedules Processes/semi-coroutines until time 'until'"""
1758
1759     """Gets called once. Afterwards, co-routines (generators) return by
1760     'yield' with a cargo:
1761     yield hold, self, <delay>: schedules the "self" process for activation
1762     after <delay> time units.If <,delay> missing,
1763     same as "yield hold,self,0"
1764
1765     yield passivate,self : makes the "self" process wait to be re-activated
1766
1767     yield request,self,<Resource>,<priority>: request 1 unit from <Resource>
1768     with <priority> pos integer (default=0)
1769
1770     yield release,self,<Resource> : release 1 unit to <Resource>
1771
1772     yield waitevent,self,<SimEvent>| [<Evt1>,<Evt2>,<Evt3>, . . . ]:
1773     wait for one or more of several events
1774
1775
1776     yield queueevent,self,<SimEvent>| [<Evt1>,<Evt2>,<Evt3>, . . . ]:
1777     queue for one or more of several events
1778
1779     yield waituntil,self,cond : wait for arbitrary condition
1780
1781     yield get,self,<buffer>,<WhatToGet>,<priority>]
1782     get <WhatToGet> items from buffer (default=1);
1783     <WhatToGet> can be a pos integer or a filter function
1784     (Store only)
1785
1786     yield put,self,<buffer>,<WhatToPut>,<priority>]
1787     put <WhatToPut> items into buffer (default=1);
1788     <WhatToPut> can be a pos integer (Level) or a list of objects
1789     (Store)
1790
1791     EXTENSIONS:
1792     Request with timeout renegeing:
1793     yield (request,self,<Resource>),(hold,self,<patience>) :
1794     requests 1 unit from <Resource>. If unit not acquired in time period
1795     <patience>, self leaves waitQ (reneges).
1796
1797     Request with event-based renegeing:
1798     yield (request,self,<Resource>),(waitevent,self,<eventlist>):
1799     requests 1 unit from <Resource>. If one of the events in <eventlist> occurs before unit
1800     acquired, self leaves waitQ (reneges).
1801
1802     Get with timeout renegeing (for Store and Level):
1803     yield (get,self,<buffer>,nrToGet etc.),(hold,self,<patience>)
1804     requests <nrToGet> items/units from <buffer>. If not acquired <nrToGet> in time period
1805     <patience>, self leaves <buffer>.getQ (reneges).
1806
1807     Get with event-based renegeing (for Store and Level):
1808     yield (get,self,<buffer>,nrToGet etc.),(waitevent,self,<eventlist>)
1809     requests <nrToGet> items/units from <buffer>. If not acquired <nrToGet> before one of
1810     the events in <eventlist> occurs, self leaves <buffer>.getQ (reneges).
1811
1812
1813

```

```

1814 Event notices get posted in event-list by scheduler after "yield" or by
1815 "activate"/"reactivate" functions.
1816
1817 """
1818 global _endtime,_e,_stop,_t,_wustep
1819 _stop=False
1820
1821 if _e is None:
1822     raise FatalSimerror("Simulation not initialized")
1823 if _e._isEmpty():
1824     message="SimPy: No activities scheduled"
1825     return message
1826
1827 _endtime=until
1828 message="SimPy: Normal exit"
1829 dispatch={hold:holdfunc,request:requestfunc,release:releasefunc,
1830           passivate:passivatefunc,waitevent:waitevfunc,queueevent:queuevfunc,
1831           waituntil:waituntilfunc,get:getfunc,put:putfunc}
1832 commandcodes=dispatch.keys()
1833 commandwords={hold:"hold",request:"request",release:"release",passivate:"passivate",
1834              waitevent:"waitevent",queueevent:"queueevent",waituntil:"waituntil",
1835              get:"get",put:"put"}
1836 nextev=_e._nextev ## just a timesaver
1837 while not _stop and _t<=_endtime:
1838     try:
1839         a=nextev()
1840         if not a[0] is None:
1841             ## 'a' is tuple "<yield command>, <action>"
1842             if type(a[0][0])==tuple:
1843                 ##allowing for yield (request,self,res),(waituntil,self,cond)
1844                 command=a[0][0][0]
1845             else:
1846                 command = a[0][0]
1847             if __debug__:
1848                 if not command in commandcodes:
1849                     raise FatalSimerror("Illegal command: yield %s"%command)
1850             dispatch[command](a)
1851         except FatalSimerror,error:
1852             print "SimPy: "+error.value
1853             sys.exit(1)
1854         except Simerror,error:
1855             message="SimPy: "+error.value
1856             _stop = True
1857         if _wustep:
1858             _test()
1859     _stopWUStepping()
1860     _e=None
1861     return message

```