Introduction to SimPy Internals

Norm Matloff

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

Contents

1	Purp	pose	1
2	Pyth	non Generators	2
3	How	y SimPy Works	4
	3.1	Running Example	4
	3.2	How initialize() Works	4
	3.3	How activate() Works	4
	3.4	How simulate() Works	5
		3.4.1 The Core while Loop	5
		3.4.2 Call to _nextev()	5
		3.4.3 How a New Event Gets Added to the Event List	7
	3.5	How Resource(), yield request and yield release Work	7
A	Sim	Py Source Code	9

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:

```
# yieldex.py example of yield, return in generator functions
2
3
   def gy():
     x = 2
4
      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()
     print g.next() # prints x, y and x+y
14
    print g.next() \# prints z/x
15
    print g.next() # causes the exception
16
17
18 if __name__ == '__main__':
      main()
19
   % python yieldex.py
   (2, 3, 5)
3
4
   Traceback (most recent call last):
    File "yieldex.py", line 19, in ?
    File "yieldex.py", line 16, in main
8
      print q.next()
  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 \mathbf{g} . Again, to learn the details on iterators, you can read my tutorial above, but all you need to know is that \mathbf{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 $\mathbf{gy}()$. Note carefully that $\mathbf{gy}()$ 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). 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 \mathbf{x} and \mathbf{y} , will be retained; for instance, \mathbf{y} will still be

• 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.

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

²The 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())
```

activate (XInst, XInst.Run())

The definition of activate() begins with

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

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 vield. 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) = hq.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** subclases, 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 \mathbf{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 $\mathbf{now}()$, 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])
```

(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.

```
#!/usr/bin/env python
       from SimPy.Lister import \star
      import heapq as hq
       import types
      import sys
      import new
      import inspect
      # $Revision: 1.1.1.75 $ $Date: 2007/12/18 13:30:47 $ kgm
"""Simulation 1.9 Implements SimPy Processes, Resources, Buffers, and the backbone simulation
scheduling by coroutine calls. Provides data collection through classes
13
      Monitor and Tally.
      Based on generators (Python 2.3 and later) """
16
17
      # yield keywords
      hold=1
      passivate=2
      request=3
       release=4
      waitevent=5
      queueevent=6
       waituntil=7
      aet=8
26
27
28
      put=9
      _endtime=0
     _t=0
_e=None
       _wustep=False #controls per event stepping for waituntil construct; not user API
         True, False
      except NameError:
      True, False = (1 == 1), (0 == 1) condQ=[]
38
39
      allMonitors=[]
allTallies=[]
      def initialize():
          global _e,_t,_stop,condQ,allMonitors,allTallies
          _____Evlist()
_t=0
43
44
            _stop=False
            allMonitors=[]
48
49
50
51
52
53
           allTallies=[]
      def now():
           return _t
      def stopSimulation():
    """Application function to stop simulation run"""
54
55
56
57
58
59
           global _stop
            _stop=True
      def _startWUStepping():
    """Application function to start stepping through simulation for waituntil construct."""
           global _wustep
61
            _wustep=True
63
64
      \begin{tabular}{ll} $\tt def \_stopWUStepping(): \\ & \tt """Application function to stop stepping through simulation.""" \\ \end{tabular}
            global _wustep
```

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

```
158
159
           def passivate(self.a):
160
                a[0][1]._nextTime=None
161
           def interrupt(self, victim):
162
163
                """Application function to interrupt active processes"""
               # can't interrupt terminated/passive/interrupted process
if victim.active():
164
165
166
                    victim.interruptCause=self # self causes interrupt
                    left=victim._nextTime-_t
victim.interruptLeft=left
167
168
                                                   # 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
                'interrupted' state.
178
179
180
               self._inInterrupt= False
181
182
           def acquired(self,res):
183
                """Multi-functional test for reneging for 'request' and 'get':
               (1) If res of type Resource:
Tests whether resource res was acquired when proces reactivated.
184
185
186
                    If yes, the parallel wakeup process is killed.
187
                    If not, process is removed from res.waitO (reneging).
188
                (2) If res of type Store:
                    Tests whether item(s) gotten from Store res. If yes, the parallel wakeup process is killed.
189
190
191
                    If no, process is removed from res.getQ
               (3) If res of type Level:

Tests whether units gotten from Level res.
192
193
194
195
                    If yes, the parallel wakeup process is killed.
                    If no, process is removed from res.getQ.
196
197
198
               if isinstance(res,Resource):
                    test=self in res.activeO
                    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
               elif isinstance(res.Store):
206
                    test=len(self.got)
208
                    if test:
                        self.cancel(self._holder)
209
210
                    else:
211
                        res.getQ.remove(self)
212
                        if res.monitored:
213
                             res.getQMon.observe(len(res.getQ),t=now())
214
                    return test
               elif isinstance(res, Level):
215
216
                    test=not (self.got is None)
217
                    if test:
218
                        self.cancel(self._holder)
219
                        res.getO.remove(self)
220
221
                        if res.monitored:
222
                             res.getQMon.observe(len(res.getQ),t=now())
223
                    return test
224
           def stored(self,buffer):
    """Test for reneging for 'yield put . . .' compound statement (Level and
Store. Returns True if not reneged.
225
226
227
               If self not in buffer.putQ, kill wakeup process, else take self out of buffer.putQ (reneged)"""
228
229
230
                test=self in buffer.putQ
231
               if test: #reneged
232
                    buffer.putQ.remove(self)
233
                    if buffer.monitored:
234
                        buffer.putQMon.observe(len(buffer.putQ),t=now())
               else:
236
                    self.cancel(self._holder)
237
               return not test
238
239
      def allEventNotices():
240
           """Returns string with eventlist as;
                    t1: processname, processname2
242
                    t2: processname4,processname5, . . .
243
244
           ...
           ret=""
245
246
           tempList=[]
247
           tempList[:]=_e.timestamps
248
           tempList.sort()
           # return only event notices which are not cancelled
```

```
250
            \label{eq:list} $$ \mathsf{tempList=}[[x[0],x[2].name]$ for $x$ in $\mathsf{tempList}$ if not $x[3]]$ $$
251
            tprev=-1
252
            for t in tempList:
253
                 # if new time, new line
if t[0] == tprev:
254
255
                     # continue line
ret+=",%s"%t[1]
256
257
                 else:
258
259
                      # new time
                     if tprev==-1:
ret="%s: %s"%(t[0],t[1])
260
261
262
                      else:
                          ret+="\n%s: %s"%(t[0],t[1])
263
                     tprev=t[0]
264
            return ret+"\n"
265
       def allEventTimes():
             """Returns list of all times for which events are scheduled.
267
268
            ....
269
270
            r[:]=_e.timestamps
            r.sort()
271
272
            \ensuremath{\text{\#}} return only event times of not cancelled event notices
273
            r1=[x[0] \text{ for } x \text{ in } r \text{ if not } r[3]]
274
            tprev=-1
275
            ret=[]
276
            for t in r1:
                 if t==tprev:
278
                     #skip time, already in list
279
                     pass
280
281
                     ret.append(t)
282
                     tprev=t
283
284
       285
286
           def __init__(self):
    # always sorted list of events (sorted by time, priority)
287
288
289
290
                 # make heapq
                 self.timestamps = []
291
                 self.sortpr=0
292
            def _post(self, what, at, prior=False):
    """Post an event notice for process what for time at"""
293
294
295
                 # event notices are Process instances
296
                 if at < t:
                     raise Simerror("Attempt to schedule event in the past")
297
                 what._nextTime = at
self.sortpr-=1
298
300
                 if prior:
                      # before all other event notices at this time
# heappush with highest priority value so far (negative of monotonely increasing number)
301
                     # store event notice in process instance
what._rec=[at,self.sortpr,what,False]
# make event list refer to it
303
304
305
306
                     hq.heappush(self.timestamps,what._rec)
307
                 else:
308
                      # heappush with lowest priority
309
                     # store event notice in process instance
what._rec=[at,-self.sortpr,what,False]
310
311
                      # make event list refer to it
                      hq.heappush(self.timestamps,what._rec)
312
313
314
            def _unpost(self, whom):
315
316
                 Mark event notice for whom as cancelled if whom is a suspended process
317
                 if whom._nextTime is not None: # check if whom was actually active
318
319
                      whom._rec[3]=True ## Mark as cancelled
320
                      whom. nextTime=None
321
            def _nextev(self):
    """Retrieve next event from event list"""
322
323
                 global _t, _stop
noActiveNotice=True
324
325
326
                 ## Find next event notice which is not marked cancelled
                 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)
                  t=_tnotice
334
                 if _t > _endtime:
   _t = _endtime
335
336
337
                      _stop = True
338
                     return (None,)
339
                     resultTuple = nextEvent._nextpoint.next()
340
341
                 except StopIteration:
```

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

```
434
                 res.append("\nNumber of observations: %s"%nrObs)
435
                 if nrObs:
436
                      su=self._sum
437
                      cum=histo[0][1]
                      fmt="%s"
438
                      439
440
                      %("%s","%s",fmt,"%s","%s","%5.1f","%s")
l1width=len(("%s <= "%fmt)%histo[1][0])
442
443
                      res.append(line1\
445
                                   %(" "*11width,ylab,histo[1][0],str(histo[0][1]).rjust(width),\
str(cum).rjust(width),(float(cum)/nrObs)*100,"%")
446
447
448
                      for i in range (1, len(histo)-1):
449
                           cum+=histo[i][1]
450
                           res.append(line)
                                    %(histo[i][0],ylab,histo[i+1][0],str(histo[i][1]).rjust(width),\
str(cum).rjust(width),(float(cum)/nr0bs)*100,"%")
451
452
453
                      \verb|cum+=histo[-1][1]|
454
                      455
456
457
                      res.append(linen\
                                    (IIIIen\
% (histo[-1][0], ylab, " "*lnwidth, str(histo[-1][1]).rjust(width), \
str(cum).rjust(width), (float(cum)/nrObs)*100, "%")
459
460
461
462
                 res.append("\n>")
return " ".join(res)
463
464
465
       def startCollection(when=0.0,monitors=None,tallies=None):
             """Starts data collection of all designated Monitor and Tally objects
466
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
474
                           m.reset()
                      for t in tallies:
    t.reset()
475
476
                      vield hold, self
            if monitors is None:
477
478
                 monitors=allMonitors
            if tallies is None:
479
                 tallies=allTallies
480
481
            s=Starter()
            activate(s,s.collect(monitors=monitors,tallies=tallies),at=when)
482
483
       class Monitor(list):
    """ Monitored variables
484
485
486
            A Class for monitored variables, that is, variables that allow one to gather simple statistics. A Monitor is a subclass of list and list operations can be performed on it. An object is established
487
488
489
490
            using m= Monitor(name = '\ldots'). It can be given a unique name for use in debugging and in tracing and ylab and tlab
491
            strings for labelling graphs.
492
493
494
                   _init__(self,name='a_Monitor',ylab='y',tlab='t'):
            def
495
                 list.__init__(self)
496
                 self.startTime = 0.0
                 self.name = name
self.ylab = ylab
self.tlab = tlab
498
499
500
                 allMonitors.append(self)
501
502
            def setHistogram(self, name = '', low=0.0, high=100.0, nbins=10):
503
                  """Sets histogram parameters.
                 Must be called before call to getHistogram"""
504
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
            def observe(self,y,t=None):
512
                 """record y and t"""
if t is None: t = now()
513
514
                 self.append([t,y])
515
            def tally(self,y):
516
                 """ deprecated: tally for backward compatibility"""
                 self.observe(y,0)
518
519
520
            def accum(self,y,t=None):
                 """ deprecated: accum for backward compatibility""" self.observe(y,t)
521
522
523
524
            def reset (self.t=None):
                  """reset the sums and counts for the monitored variable """
```

```
self[:]=[]
if t is None: t = now()
526
527
528
                  self.startTime = t
529
530
            def tseries(self):
531
                   """ the series of measured times"""
                  return list(zip(*self)[0])
532
            def yseries(self):
    """ the series of measured values"""
    return list(zip(*self)[1])
534
535
537
538
            def count(self):
539
                  """ deprecated: the number of observations made """
540
                  return self.__len__()
541
            def total(self):
    """ the sum of the y"""
    if self.__len__()==0: return 0
543
545
546
                       sum = 0.0
                       for i in range(self.__len__()):
                       sum += self[i][1]
return sum # replace by sum() later
548
549
            def mean(self):
    """ the simple average of the monitored variable"""
    try: return 1.0*self.total()/self.__len__()
551
552
553
554
                  except: print 'SimPy: No observations for mean'
555
556
                  """ the sample variance of the monitored variable """ n = len(self)
557
559
                  tot = self.total()
                  ssq=0.0
560
                 ssq=0.0
##yy = self.yseries()
for i in range(self.__len__()):
    ssq += self[i][1]**2 # replace by sum() eventually
try: return (ssq - float(tot*tot)/n)/n
561
562
563
565
566
                  except: print 'SimPy: No observations for sample variance'
            568
569
570
                       If t is used it is assumed to be the current time,
                  otherwise t = now()
571
572
573
                  N = self.__len__()
                  if N == 0:
    print 'SimPy: No observations for timeAverage'
574
575
576
                       return None
577
578
                  if t is None: t = now()
                  sum = 0.0
tlast = self.startTime
#print 'DEBUG: timave ',t,tlast
579
580
581
582
                  ylast = 0.0
for i in range(N):
583
                       ti,yi = self[i]
sum += ylast*(ti-tlast)
tlast = ti
584
585
586
                  ylast = yi

sum += ylast*(t-tlast)

T = t - self.startTime

if T == 0:
587
588
589
590
                        print 'SimPy: No elapsed time for timeAverage'
591
                        return None
593
                  #print 'DEBUG: timave ',sum,t,T
594
                  return sum/float(T)
595
596
            def timeVariance(self, t=None):
597
                    "" the time-weighted Variance of the monitored variable.
                 If t is used it is assumed to be the current time, otherwise t = now()
598
599
601
                  N = self.___1
if N == 0:
602
                               _len__()
603
                       print 'SimPy: No observations for timeVariance' return None
604
605
606
                  if t is None: t = now()
                  sm = 0.0
ssq = 0.0
607
608
                  tlast = self.startTime
# print 'DEBUG: 1 twVar ',t,tlast
609
610
                  ylast = 0.0
611
612
                  for i in range(N):
613
                       ti, yi = self[i]
                       sm += ylast*(ti-tlast)
614
615
                       ssq += ylast*ylast*(ti-tlast)
                       tlast = ti
616
617
                       ylast = yi
```

```
618
                  sm += vlast*(t-tlast)
                 sm +- ylast*(t-tlast)
ssq += ylast*(t-tlast)
T = t - self.startTime
if T == 0:
    print 'SimPy: No elapsed time for timeVariance'
619
620
621
622
623
                 mn = sm/float(T)
# print 'DEBUG: 2 twVar ',ssq,t,T
624
625
626
                  return ssq/float(T) - mn*mn
627
628
629
            def histogram(self,low=0.0,high=100.0,nbins=10):
    """ A histogram of the monitored y data values.
630
                 ....
631
632
                 h = Histogram(name=self.name,low=low,high=high,nbins=nbins)
633
                  vs = self.yseries()
634
                  for y in ys: h.addIn(y)
635
                 return h
636
            def getHistogram(self):
    """Returns a histogram based on the parameters provided in
    preceding call to setHistogram.
637
638
639
640
                 ys = self.yseries()
641
642
                  h=self.histo
643
                  for y in ys: h.addIn(y)
644
                 return h
645
646
            def printHistogram(self,fmt="%s"):
    """Returns formatted frequency distribution table string from Monitor.
647
648
                  Precondition: setHistogram must have been called.
                  fmt==format of bin range values
649
650
651
652
                       histo=self.getHistogram()
653
                  except:
654
655
                       656
                 ylab=self.ylab
657
658
                 nrObs=self.count()
width=len(str(nrObs))
659
                  res=[]
                 res_ipend("\nHistogram for %s:"%histo.name)
res.append("\nNumber of observations: %s"%nrObs)
660
661
662
                  su=sum(self.yseries())
663
                  cum=histo[0][1]
                 cum-histo[U][1]
line="\n%s <= %s < %s: %s (cum: %s/%s%s)"\
    %(fmt, "%s", fmt, "%s", "%s", "%5.1f", "%s")
line1="\n%s%s < %s: %s (cum: %s/%s%s)"\
    %("%s", "%s", fmt, "%s", "%s", "%5.1f", "%s")
llwidth=len(("%s <= "%fmt)%histo[1][0])</pre>
664
665
666
667
668
669
                  res.append(line1\
670
                                %(" "*11width,ylab,histo[1][0],str(histo[0][1]).rjust(width),\
671
672
                                 str(cum).rjust(width),(float(cum)/nrObs)*100,"%")
673
                  for i in range(1,len(histo)-1):
674
                       cum+=histo[i][1]
675
                       res.append(line\
676
                                %(histo[i][0],ylab,histo[i+1][0],str(histo[i][1]).rjust(width),
677
                                   str(cum).rjust(width),(float(cum)/nrObs)*100,"%")
678
679
                  cum+=histo[-1][1]
                 680
681
682
683
                  res.append(linen\
                                %(histo[-1][0],ylab," "*lnwidth,str(histo[-1][1]).rjust(width),\
684
685
                                str(cum).rjust(width),(float(cum)/nrObs)*100,"%"
686
                  return " ".join(res)
687
688
689
       class Tallv:
            def __init__(self, name="a_Tally", ylab="y",tlab="t"):
690
                 self.name = name
self.ylab = ylab
691
692
693
                  self.tlab = tlab
694
                  self.reset()
695
                  self.startTime = 0.0
696
                 self.histo = None
self.sum = 0.0
697
698
                  self._sum_of_squares = 0
                 self._integral = 0.0  # time-weighted sum
self._integral2 = 0.0  # time-weighted sum of squares
699
700
701
                  allTallies.append(self)
702
            def setHistogram(self, name = '', low=0.0, high=100.0, nbins=10):
703
                 """sets histogram parameters.

Must be called to prior to observations initiate data collection
704
705
706
                  for histogram.
707
                 if name=='':
708
709
                       hname=self.name
```

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

```
802
                   except:
                         raise FatalSimerror("histogramTable: call setHistogram first"\
" for Tally %s"%self.name)
803
804
                   ylab=self.ylab
805
806
                   nrObs=self.count()
807
                    width=len(str(nrObs))
808
                   res=[]
                   res.append("\nHistogram for %s:"%histo.name)
res.append("\nNumber of observations: %s"%nrObs)
809
810
811
                   su=self.total()
                   cum=histo[0][1]
812
                   line="\n\s <= \s < \s: \s (cum: \s/\s\s)\\
\s (fmt,"\s",fmt,"\s","\s","\s.1f","\s")
line1="\n\s\s < \s: \s (cum: \s/\s\s\s)"\
813
814
                   %("%s","%s",fmt,"%s","%s","%5.1f","%s")
l1width=len(("%s <= "%fmt)%histo[1][0])
816
817
                   res.append(line1\
%(" "*llwidth,ylab,histo[1][0],str(histo[0][1]).rjust(width),\
str(cum).rjust(width),(float(cum)/nr0bs)*100,"%")
819
820
821
                   for i in range(1,len(histo)-1):
    cum+=histo[i][1]
822
823
824
                         res.append(line\
                                   % (histo[i][0],ylab,histo[i+1][0],str(histo[i][1]).rjust(width),\
str(cum).rjust(width),(float(cum)/nrObs)*100,"%")
825
826
827
                   cum+=histo[-1][1]
linen="\n%s <= %s %s : %s (cum: %s/%s%s)"\
   %(fmt, "%s", "%s", "%s", "%s", "%5.1f", "%s")
lnwidth=len(("<%s"%fmt, whisto[1][0])</pre>
828
829
830
831
832
                   res.append(linen\
                                   (linen\
%(histo[-1][0],ylab," "*lnwidth,str(histo[-1][1]).rjust(width),\
str(cum).rjust(width),(float(cum)/nrObs)*100,"%")
833
834
835
                   return " ".join(res)
836
837
838
        class Queue(list):
             def __init__(self,res,moni):
    if not moni is None: #moni==[]:
839
841
                         self.monit=True # True if a type of Monitor/Tally attached
                   else:
842
843
                         self.monit=False
                   self.moni=moni # The Monitor/Tally
self.resource=res # the resource/buffer this queue belongs to
844
845
846
             def enter(self,obj):
847
848
                   pass
849
             def leave(self):
850
                   pass
852
853
             def takeout(self.obi):
854
                   self.remove(obj)
855
                   if self.monit:
                         self.moni.observe(len(self),t=now())
856
857
858
        class FIFO(Queue):
             def __init__(self,res,moni):
    Queue.__init__(self,res,moni)
859
860
861
862
             def enter(self,obj):
863
                   self.append(obj)
864
                   if self.monit:
865
                         self.moni.observe(len(self),t=now())
866
867
             def enterGet (self.obi):
868
                   self.enter(obj)
869
             def enterPut(self,obj):
870
871
                   self.enter(obj)
872
873
             def leave(self):
874
                   a= self.pop(0)
875
                   if self.monit:
876
                         self.moni.observe(len(self),t=now())
877
                   return a
878
        class PriorityQ(FIFO):
880
             """Queue is always ordered according to priority.
Higher value of priority attribute == higher priority.
881
             def __init__(self,res,moni):
    FIFO.__init__(self,res,moni)
883
884
885
             def enter(self,obj):
    """Handles request queue for Resource"""
886
887
888
                   if len(self):
889
                         ix=self.resource
                         if self[-1]._priority[ix] >= obj._priority[ix]:
890
891
                               self.append(obj)
                         else:
892
893
                              z=0
```

```
894
                          while self[z]._priority[ix] >= obj._priority[ix]:
895
                          self.insert(z,obj)
897
                else:
                    self.append(obi)
898
899
                if self.monit:
                     self.moni.observe(len(self),t=now())
900
901
902
           def enterGet(self,obj):
                """Handles getQ in Buffer"""
if len(self):
903
                     ix=self.resource
#print "priority:",[x._priority[ix] for x in self]
if self[-1]._getpriority[ix] >= obj._getpriority[ix]:
905
906
908
                          self.append(obj)
909
911
                          while self[z]._getpriority[ix] >= obj._getpriority[ix]:
912
913
                          self.insert(z,obj)
914
                else:
915
                     self.append(obj)
916
                if self.monit:
                     self.moni.observe(len(self),t=now())
917
           def enterPut(self,obj):
    """Handles putQ in Buffer"""
919
920
                     ix=self.resource
#print "priority:",[x._priority[ix] for x in self]
if self[-1]._putpriority[ix] >= obj._putpriority[ix]:
922
923
924
925
                          self.append(obj)
                     else:
926
927
928
                          while self[z]._putpriority[ix] >= obj._putpriority[ix]:
929
930
                          self.insert(z,obj)
931
                else:
                    self.append(obj)
933
                if self.monit:
                     self.moni.observe(len(self),t=now())
934
936
       class Resource(Lister):
            """Models shared, limited capacity resources with queuing;
937
           FIFO is default queuing discipline.
938
939
940
941
           def __init__(self,capacity=1,name="a_resource",unitName="units",
942
                           \verb|qType=FIFO|, \verb|preemptable=0|, \verb|monitored=False|, \verb|monitorType=Monitor||:
                monitorType={Monitor(default) | Tally}
"""
944
945
                self.name=name
                                             # resource name
                self.capacity=capacity # resource units in this resource
self.unitName=unitName # type name of resource units
947
948
949
                self.n=capacity
                                            # uncommitted resource units
950
                self.monitored=monitored
951
952
953
                if self.monitored:
                                                  # Monitor waitQ, activeQ
                    self.actMon=monitorType(name="Active Queue Monitor %s"%self.name, ylab="nr in queue",tlab="time")
954
955
956
                     monact=self.actMon
                     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):
    """Process request event for this resource"""
968
969
                obj=arg[1]
                    if len(arg[0]) == 4:
970
972
973
                if self.preemptable and self.n == 0: \# No free resource
974
                     # test for preemption condition
preempt=obj._priority[self] > self.activeQ[-1]._priority[self]
975
976
                     # If yes:
978
                     if preempt:
979
                         z=self.activeQ[-1]
980
                          # suspend lowest priority process being served
981
                          \#\#suspended = z
                          # record remaining service time
982
983
                          z._remainService = z._nextTime - _t
                          Process().cancel(z)
984
985
                          # remove from activeQ
```

```
986
                         self.activeQ.remove(z)
987
                         # put into front of wait0
988
                         self.waitQ.insert(0,z)
989
                         # if self is monitored, update wait0 monitor
990
                         if self.monitored:
991
992
                             self.waitMon.observe(len(self.waitQ),now())
                        \# record that it has been preempted z._preempted = 1
993
994
995
                         # passivate re-queued process
                        z._nextTime=None
# assign resource unit to preemptor
996
997
998
                         self.activeQ.enter(obj)
                         # post event notice for preempting process
999
                         _e._post(obj,at=_t,prior=1)
1000
                    else:
                        self.waitQ.enter(obj)
1001
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
                        obj._nextTime=None
1008
1009
                    else:
                        self.n -= 1
1010
1011
                        self.activeQ.enter(obj)
1012
                         _e._post(obj,at=_t,prior=1)
1014
           def _release(self,arg):
    """Process release request for this resource"""
1015
1016
                self.n += 1
1017
                self.activeQ.remove(arg[1])
                if self.monitored:
1018
1019
                    self.actMon.observe(len(self.activeQ),t=now())
1020
                #reactivate first waiting requestor if any; assign Resource to it
                if self.waitQ:
1022
                    obj=self.waitQ.leave()
1023
                                              #assign 1 resource unit to object
                    self.n -= 1
                    self.activeQ.enter(obj)
1024
1025
                    # if resource preemptable:
1026
                    if self.preemptable:
1027
                         # if object had been preempted:
1028
                        if obj._preempted:
                             boj._preempted = 0
# reactivate object delay= remaining service time
1029
1030
1031
                             \verb|reactivate(obj,delay=obj.\_remainService)| \\
1032
                         # else reactivate right away
1033
                             reactivate(obj,delay=0,prior=1)
1034
1035
1036
               reactivate(obj,delay=0,prior=1)
_e._post(arg[1],at=_t,prior=1)
1037
1038
1039
       class Buffer(Lister):
1040
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",
                        put(Type=FIFO, get(Type=FIFO, monitord=False, monitorType=Monitor, initialBuffered=None):
1048
1049
1050
                if capacity=="unbounded": capacity=sys.maxint
                self.capacity=capacity
1051
                self.name=name
1052
1053
                self.putQType=putQType
1054
               self.getQType=getQType
self.monitored=monitored
1055
1056
                self.initialBuffered=initialBuffered
1057
                self.unitName=unitName
1058
                if self.monitored:
                    \#\# monitor for Producer processes' queue
1059
                    1060
1061
1062
                    ylab="nr in queue",tlab="time") ## monitor for nr items in buffer
1064
1065
                    self.bufferMon=monitorType(name="Buffer Monitor %s"%self.name,
1066
1067
                                              ylab="nr in buffer",tlab="time")
1068
                else:
1069
                    self.putQMon=None
                    self.getQMon=None
self.bufferMon=None
1070
1071
1072
                self.putQ=self.putQType(res=self,moni=self.putQMon)
1073
                \verb|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())
                self._putpriority={}
```

```
1078
                self._getpriority={}
1079
1080
                def _put(self):
                pass
def _get(self):
pass
1081
1082
1083
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
            def __init__(self,**pars):
1096
1097
                Buffer.__init__(self,**pars)
if self.name is None:
1098
                     self.name="a_level"
1099
                                             ## default name
1100
                if (type(self.capacity)!=type(1.0) and\
1101
                         type(self.capacity)!=type(1)) or\
1102
1103
                          self.capacity<0:
1104
                         raise FatalSimerror\
                              ("Level: capacity parameter not a positive number: s"
1106
                              %self.initialBuffered)
1107
1108
                if type(self.initialBuffered) == type(1.0) or \setminus
                     type(self.initialBuffered) == type(1):
if self.initialBuffered>self.capacity:
1109
1110
1111
                          raise FatalSimerror("initialBuffered exceeds capacity")
                     if self initialBuffered>=0.
1112
                         self.nrBuffered=self.initialBuffered ## nr items initially in buffer
1113
1114
                                                     ## buffer is just a counter (int type)
1115
                     else:
                         raise FatalSimerror\
1116
1117
                          ("initial
Buffered param of Level negative: s"\
1118
                          %self.initialBuffered)
                elif self.initialBuffered is None:
1120
                     self.initialBuffered=0
1121
                     self.nrBuffered=0
1122
1123
                     raise FatalSimerror\
                          ("Level: wrong type of initialBuffered (parameter=%s)"\%self.initialBuffered)
1124
1125
1126
                if self.monitored:
                     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"""
                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]
                elif len(arg[0]) == 4: # yield get,self,buff,whattoput
obj_putpriority[self]=Buffer.priorityDefault #default
1136
1137
1138
                     whatToPut=arg[0][3]
                     # yield get,self,buff
obj._putpriority[self]=Buffer.priorityDefault #default
1139
1140
1141
                     whatToPut=1
1142
                if type(whatToPut)!=type(1) and type(whatToPut)!=type(1.0):
1143
                     raise FatalSimerror("Level: put parameter not a number")
                if not whatToPut>=0.0:
1144
1145
                     raise FatalSimerror("Level: put parameter not positive number")
                whatToPutNr=whatToPut
1146
1147
                if whatToPutNr+self.amount>self.capacity:
                     obj._nextTime=None #pa
obj._whatToPut=whatToPutNr
1148
                                            #passivate put requestor
1149
1150
                     self.putQ.enterPut(obj)
                                                  #and queue, with size of put
1151
                else:
                     self.nrBuffered+=whatToPutNr
1152
1153
                     if self.monitored:
1154
                         self.bufferMon.observe(y=self.amount,t=now())
                     # service any getters waiting
1156
                     # service in queue-order; do not serve second in queue before first
1157
                     # has been served
                     while len(self.getQ) and self.amount>0:
1158
1159
                         proc=self.getQ[0]
                          if proc._nrToGet<=self.amount:
1160
                              proc.got=proc._nrToGet
1162
                              self.nrBuffered-=proc.got
1163
                              if self.monitored:
                                  self.bufferMon.observe(y=self.amount,t=now())
1164
                              self.getQ.takeout(proc) # get requestor's record out of queue
_e._post(proc,at=_t) # continue a blocked get requestor
1165
1166
1167
                         else:
1168
                              break
1169
                     _e._post(obj,at=_t,prior=1) # continue the put requestor
```

```
1170
           def _get(self,arg):
    """Handles get requests for Level instances"""
1171
1172
                obj=arg[1]
1173
1174
                obi.got=None
1175
                 if len(arg[0]) == 5:
                                                # yield get, self, buff, whattoget, priority
                     obj._getpriority[self]=arg[0][4]
nrToGet=arg[0][3]
1176
                    r len(arg[0]) == 4:  # yield get, self, buff, whattoget
obj._getpriority[self] = Buffer.priorityDefault #default
nrToGet=arg[0][3]
a:
1178
                 elif len(arg[0]) == 4:
1179
1181
                else.
                     # yield get,self,buff
obj._getpriority[self]=Buffer.priorityDefault
1182
1183
                     nrToGet=1
1184
                if type(nrToGet)!=type(1.0) and type(nrToGet)!=type(1):
1185
                     raise FatalSimerror\
                         ("Level: get parameter not a number: %s"%nrToGet)
1186
1187
                if nrToGet<0:
                     raise FatalSimerror\
1188
1189
                         ("Level: get parameter not positive number: %s"%nrToGet)
                if self.amount < nrToGet:
1190
1191
                     obj._nrToGet=nrToGet
1192
                     self.getQ.enterGet(obj)
1193
                     # passivate queuing process
1194
                     obj._nextTime=None
1195
                else.
1196
                     obj.got=nrToGet
                     self.nrBuffered-=nrToGet
1198
                     if self.monitored:
1199
                         self.bufferMon.observe(y=self.amount,t=now())
1200
                      _e._post(obj,at=_t,prior=1)
                     # reactivate any put requestors for which space is now available
# service in queue-order; do not serve second in queue before first
1201
1202
1203
                     # has been served
                     while len(self.putQ): #test for queued producers
1204
1205
                         proc=self.putQ[0]
1206
                          if proc._whatToPut+self.amount<=self.capacity:</pre>
1207
                              self.nrBuffered+=proc._whatToPut
                              if self.monitored:
1208
                                   self.bufferMon.observe(y=self.amount,t=now())
1209
                              self.putQ.takeout(proc)#requestor's record out of queue
1210
                              _e._post(proc,at=_t) # continue a blocked put requestor
1212
                          else.
1213
                              break
1214
1215
       class Store (Buffer):
            """Models buffers for processes coupled by putting/getting distinguishable
1216
1217
            Blocks a process when a put would cause buffer overflow or a get would cause
1218
            buffer underflow.
            Default queuing discipline for blocked processes is priority FIFO.
1220
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
                 _init__(self,**pars):
1231
                 Buffer.__init__(self,**pars)
                self.theBuffer=[]
1232
1233
                if self.name is None:
                     self.name="a_store" ## default name
1234
                if type(self.capacity)!=type(1) or self.capacity<=0:
1235
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:
                         raise FatalSimerror("initialBuffered exceeds capacity")
1241
1242
                \tt self.theBuffer[:]=self.initialBuffered\#\#buffer==list\ of\ objects\ elif\ self.initialBuffered\ is\ None:
1243
1244
1245
                     self.theBuffer=[]
1246
                else:
1247
                     raise FatalSimerror\
                          ("Store: initialBuffered not a list")
1248
                if self.monitored:
1249
1250
                     self.bufferMon.observe(y=self.nrBuffered,t=now())
                self._sort=None
1251
1252
1253
1254
1255
           def addSort(self,sortFunc):
1256
                 """Adds buffer sorting to this instance of Store. It maintains
                the
Buffer sorted by \bar{\text{the}} sort
Attr attribute of the objects in the
1257
1258
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
                self._sort=new.instancemethod(sortFunc,self,self.__class__)
self.theBuffer=self._sort(self.theBuffer)
1268
1269
1270
           def _put(self,arg):
    """Handles put requests for Store instances"""
1271
1272
1273
                obj=arg[1]
                if len(arg[0]) == 5:
                                                 # vield put, self, buff, whattoput, priority
1274
1275
                     obj._putpriority[self] = arg[0][4]
1276
                     whatToPut=arg[0][3]
                elif len(arg[0]) == 4:
                                                 # yield put, self, buff, whattoput
1277
                     obj._putpriority[self]=Buffer.priorityDefault #default
1278
1279
                     whatToPut=arg[0][3]
1280
                else:
                                                 # error, whattoput missing
                     raise FatalSimerror("Item to put missing in yield put stmt")
1281
                if type(whatToPut)!=type([]):
    raise FatalSimerror("put parameter is not a list")
1282
1283
1284
                 whatToPutNr=len(whatToPut)
1285
                if whatToPutNr+self.nrBuffered>self.capacity:
                     obj._nextTime=None #passivate put requestor obj._whatToPut=whatToPut
1287
                     self.putQ.enterPut(obj) #and queue, with items to put
1288
1289
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
                     # service any waiting getters
# service in queue order: do not serve second in queue before first
1296
1297
1298
                     # has been served
                     while self.nrBuffered>0 and len(self.get0):
1299
1300
                          proc=self.getQ[0]
1301
                          if inspect.isfunction(proc._nrToGet):
                              movCand=proc._nrToGet(self.theBuffer) #predicate parameter
1302
1303
                              if movCand:
1304
                                   proc.got=movCand[:]
                                   for i in movCand:
self.theBuffer.remove(i)
1305
1306
                                   self.getQ.takeout(proc)
1307
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 else:
1312
                                  break
                          else: #numerical parameter
1313
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:]
if self.monitored:
1319
1320
                                        self.bufferMon.observe(y=self.nrBuffered,t=now())
1321
                                   # take this get requestor's record out of queue:
self.getQ.takeout(proc)
1322
1323
                                   _e._post(what=proc,at=_t) # continue a blocked get requestor
1324
                              else:
1325
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]
                                                 # the list of items retrieved by 'get'
1333
                obi.aot=[]
1334
                if len(arg[0]) == 5:
                                                 # yield get,self,buff,whattoget,priority
                     obj._getpriority[self]=arg[0][4]
1335
1336
                     if inspect.isfunction(arg[0][3]):
1337
                          filtfunc=arg[0][3]
1338
                     else:
1339
                         nrToGet=arg[0][3]
1340
                elif len(arg[0]) == 4:  # yield get,self,buff,whattoget
    obj__getpriority[self]=Buffer.priorityDefault #default
1341
                     if inspect.isfunction(arg[0][3]):
1342
1343
                          filtfunc=arg[0][3]
1344
                     else:
1345
                         nrToGet=arg[0][3]
                                                 # vield get, self, buff
1346
1347
                     obj._getpriority[self]=Buffer.priorityDefault
1348
                if not filtfunc: #number specifies nr items to get if nrToGet < 0:
1349
1350
1351
                          raise FatalSimerror\
                     ("Store: get parameter not positive number: %s"%nrToGet)
if self.nrBuffered < nrToGet:</pre>
1352
```

```
1354
                         obj._nrToGet=nrToGet
1355
                          self.getO.enterGet(obi)
1356
                          # passivate/block queuing 'get' process
                         obj._nextTime=None
1357
1358
                     else:
1359
                          for i in range(nrToGet):
                              1360
1361
1362
                         if self.monitored:
                              self.bufferMon.observe(v=self.nrBuffered,t=now())
1363
                          _e._post(obj,at=_t,prior=1)
1365
                         # reactivate any put requestors for which space is now available
# serve in queue order: do not serve second in queue before first
1366
                          # has been served
1368
                          while len(self.putQ):
1369
                              proc=self.putQ[0]
1370
                              if len(proc._whatToPut)+self.nrBuffered<=self.capacity:
                                   for i in proc._whatToPut:
    self.theBuffer.append(i) #move items to buffer
1371
1372
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())
                                   self.putQ.takeout(proc) # dequeue requestor's record
_e._post(proc,at=_t) # continue a blocked put requestor
1377
1379
                              else.
1380
                                  break
1381
                else: # items to get determined by filtfunc
1382
                     movCand=filtfunc(self.theBuffer)
1383
                     if movCand: # get succeded
1384
                          _e._post(obj,at=_t,prior=1)
                         obj.got=movCand[:]
for item in movCand:
1385
1386
1387
                              self.theBuffer.remove(item)
                         if self.monitored:
    self.bufferMon.observe(y=self.nrBuffered,t=now())
1388
1389
1390
                          # reactivate any put requestors for which space is now available
1391
                          # serve in queue order: do not serve second in queue before first
                          # has been served
1392
1393
                          while len(self.putQ):
1394
                              proc=self.putQ[0]
if len(proc._whatToPut)+self.nrBuffered<=self.capacity:</pre>
1395
1396
                                   for i in proc._whatToPut:
    self.theBuffer.append(i) #move items to buffer
1397
1398
                                   if not(self._sort is None):
                                       self.theBuffer=self._sort(self.theBuffer)
1399
1400
                                   if self.monitored:
1401
                                       self.bufferMon.observe(y=self.nrBuffered,t=now())
1402
                                   self.putQ.takeout(proc) # dequeue requestor's record
_e._post(proc,at=_t) # continue a blocked put requestor
1404
                              else:
1405
                                  break
                     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
            """Supports one-shot signalling between processes. All processes waiting for an event to occur
1413
1414
           get activated when its occurrence is signalled. From the processes queuing for an event, only
            the first gets activated.
1415
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).
                Reactivates ALL processes waiting for this event. (Cleanup waits lists
1427
                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).)
1432
                self.signalparam=param
if not self.waits and not self.queues:
1433
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
                         for ev in p[1]:
    if ev!=self:
1441
1442
1443
                                   if ev.occurred:
                                   \label{eq:p0} $p[0].eventsFired.append(ev)$ for iev in ev.waits:
1444
```

```
1446
                                        if iev[0]==p[0]:
1447
                                            ev.waits.remove(iev)
1448
                     self.waits=[]
1449
1450
                     if self.queues:
1451
                          proc=self.queues.pop(0)[0]
1452
                          proc.eventsFired.append(self)
1453
                          reactivate(proc)
1454
           def _wait(self,par):
    """Consumes a signal if it has occurred, otherwise process 'proc'
1455
1456
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:
                     self.waits.append([proc,[self]])
1462
1463
                     proc._nextTime=None #passivate calling process
1464
                else:
1465
                     proc.eventsFired.append(self)
                     self.occurred=Fals
1466
1467
                     _e._post(proc,at=_t,prior=1)
1468
           def _waitOR(self,par):
    """Handles waiting for an OR of events in a tuple/list.
1469
1470
1471
1472
                proc=par[0][1
1473
                evlist=par[0][2]
1474
                proc.eventsFired=[]
1475
                anvoccur=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=[]
                     proc._nextTime=None #passivate calling process
1486
1487
                     for ev in evlist:
1488
                          ev.waits.append([proc,evlist])
1489
1490
           def _queue(self,par):
    """Consumes a signal if it has occurred, otherwise process 'proc'
1491
1492
                queues for this event.
1493
1494
                \verb|proc=par[0][1]| \verb|#the process issuing the yield queue event command|\\
1495
                 proc.eventsFired=[]
1496
                 if not self.occurred:
                     self.queues.append([proc,[self]])
proc._nextTime=None #passivate calling process
1497
1498
1499
1500
                     proc.eventsFired.append(self)
1501
                     self.occurred=False
1502
                     _e._post(proc,at=_t,prior=1)
1503
            def _queueOR(self,par):
1504
                   "Handles queueing for an OR of events in a tuple/list.
1505
1506
                ....
1507
                proc=par[0][1]
                evlist=par[0][2]
proc.eventsFired=[]
1508
1509
1510
                 anyoccur=False
1511
                for ev in evlist:
1512
                     if ev.occurred:
1513
                          anyoccur=True
                          proc.eventsFired.append(ev)
1514
1515
                          ev.occurred=False
1516
                if anyoccur: #at least one event has fired; continue process
1517
                     _e._post(proc,at=_t,prior=1)
1518
                else: \#no event in list has fired, enter process in all 'waits' lists
1519
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
       def _test():
1526
1527
1528
           Gets called by simulate after every event, as long as there are processes
           waiting in condQ for a condition to be satisfied.

Tests the conditions for all waiting processes. Where condition satisfied, reactivates that process immediately and removes it from queue.
1529
1530
1531
1532
           global condO
1533
1534
            rList=[]
1535
            for el in condQ:
1536
                if el.cond():
                     rList.append(el)
```

```
reactivate(el)
for i in rList:
1538
1539
1540
                condQ.remove(i)
1541
1542
           if not condO:
1543
                _stopWUStepping()
1544
1545
       def _waitUntilFunc(proc,cond):
           global condQ
1546
1547
           Puts a process 'proc' waiting for a condition into a waiting queue.
1548
            'cond' is a predicate function which returns True if the condition is
1549
           satisfied.
1550
1551
1552
           if not cond():
1553
                condQ.append(proc)
1554
                proc.cond=cond
                                              #signal 'simulate' that a process is waiting
1555
                 _startWUStepping()
1556
                # passivate calling process
1557
                proc._nextTime=None
1558
           else:
                #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
              "Handles 'yield request, self, res' and 'yield (request, self, res), (<code>, self, par)'.
1575
            <code> can be 'hold' or 'waitevent'.
1576
1577
           if type(a[0][0]) == tuple:
1578
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
                ##a[1] is the Process instance
1584
1585
                b[2]._request(arg=(b,a[1]))
1586
                ##deal with add-on condition to command
1587
                ##Trigger processes for reneging
                class _Holder(Process):
    """Provides timeout process"""
1588
1589
1590
                     def trigger(self, delay):
1591
                         yield hold, self, delay
1592
                         if not proc in b[2].activeQ:
1593
                              reactivate(proc)
1594
                class _EventWait(Process):
    """Provides event waiting process"""
1595
1596
                     def trigger(self, event):
1597
1598
                         yield waitevent, self, event
1599
                          if not proc in b[2].activeQ:
                              a[1].eventsFired=self.eventsFired
1600
1601
                              reactivate(proc)
1602
1603
                #activate it
                proc=a[0][0][1] # the process to be woken up
1604
1605
                 actCode=a[0][1][0]
1606
                if actCode==hold:
1607
                     proc._holder=_Holder(name="RENEGE-hold for %s"%proc.name)
                                                                       the timeout delay
1608
                     activate(proc._holder,proc._holder.trigger(a[0][1][2]))
1609
                elif activate(plot._initer)prot._initeringset(t.,...,
elif activate(plot._initerint)
elif activate(plot._initerint)
raise FatalSimerror("Illegal code for reneging: waituntil")
1610
1611
                elif actCode==waitevent:
1612
1613
                     proc._holder=_EventWait(name="RENEGE-waitevent for %s"%proc.name)
1614
                     activate(proc._holder,proc._holder.trigger(a[0][1][2]))
1616
                elif actCode==queueevent:
    raise FatalSimerror("Illegal code for reneging: queueevent")
1617
                else:
1619
                     raise FatalSimerror("Illegal code for reneging %s"%actCode)
1620
                ## Simple yield request command
1621
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)
```

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

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

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