SimulationGUIDebug

Brian Jacobs, Kip Nicol, Logan Rockmore

January 2009

SimPy Version: 2.0

1 SimulationGUIDebug Overview

The SimulationGUIDebug SimPy package is a tool that lets users debug SimPy programs visually. The package can be used as either a standalone debugger, or as a supplement to existing debuggers such as Python's internal debugger, PDB. The package contains a number of user APIs that can be used to create windows for different objects within SimPy programs. The package contains a few specialized windows: *EventListWindow*, *ProcessWindow*s, and *ResourceWindow*s each which contain important default information. User defined hooks which return a **str** type, can be used to print out data in these windows.

2 System Requirements

SimPy 2.0

3 User Instructions

3.1 Setup

3.1.1 Registering Windows

In order for SimulationGUIDebug to create windows for your SimPy objects you must first register them. SimulationGUIDebug gives you the option to register any object or *SimPy.Process* subclasses. SimulationGUIDebug will create specialized windows for both the *SimPy.Process* and *SimyPy.Resource* instances when they are passed to **register()**. If a *SimPy.Process* subclass is passed to **register()** then SimulationGUIDebug will create *ProcessWindows* for each instance of the class automatically once the instance is activated. The **register()** function also lets the user pass in an optional *hook* function (which returns a string) to print user defined text on the window. The **register()** function also lets the user pass in an optional *name* parameter to specify the title of the window.

register

- **Call:** *register(obj[,hook,name])*
- **Parameters:** *obj*: any object or *SimPy.Process* subclass class. *hook*: a function that returns a string. *name* a string to be used as the window title.

- **Return Value:** None

3.1.2 Specifying A Window Title

It is recommended that you give names to your Resources and Processes. SimulationGUIDebug uses the name stored in the *name* variable inside the Resource and Process classes to create the titles for the windows. The default names for a Resource is a_resource and the name for a Process is a_process. To name a process, type:

```
Process.__init__(self,name="CarArrivals")
```

where CarArrivals is the name of the Process. The same can be done with a Resource.

3.1.3 Setting Run Mode

SimulationGUIDebugger runs in two modes. The first mode uses SimulationGUIDebugger's own user prompt and steps through your simulation using SimulatinGUIDebugger's own method. The drawback to this is that you can't run another debugger in parallel. If you wish to use another debugger, you need to set the run mode to **NO_STEP**. To do this, enter the following line of code to your simulation:

```
SimulationGUIDebug.setRunMode(NO_STEP)
```

This will only create the windows when you run the simulation and updates them after every event. This allows you to use another debugger and still use SimulationGUIDebugger's windows.

setRunMode

- **Call:** *setRunMode(runMode)*
- **Parameters:** *runMode*: SimulationGUIDebug.STEP, SimulationGUIDebug.NO_STEP
- **Return Value:** None

3.1.4 initialize() and simulate()

The methods of using *initialize()* and *simulate()* have not changed from their use in SimPy.

3.2 Execution/Operation

To start debugging, simply run your simulation as you normally would. SimulationGUIDebugger will take over from here and start by displaying the following menu:

```
[c]   Continue simulation
[s]   Step to next event
[b #] Add new breakpoint
[q]   Quit debugger
```

Typing c will continue simulation will run your simulation until the next breakpoint or if there are no more breakpoints, it will finish your simulation. S will run the simulation until the next event and stop; updating all windows in the process. Adding a breakpoint will add a breakpoint at a given simulated time instead of line number like most debuggers. Once a breakpoint is reached, the debugger will stop the simulation at the next event after the breakpoint. Quitting the debugger will end the simulation and close all related windows.

4 Understanding The GUI

4.1 EventWindow Class

The EventWindow Class is used to create objects that control event windows. The window contains a table with the current event list data, the current time *now*.

The window's table has three columns: *Time* and *Process*. The *Time* column contains the time at which the corresponding event is scheduled to arrive. The *Process* column contains the name of the process that is scheduled at its corresponding time. The *Next Line* column contains the line number for the next line (in code) that the corresponding process will execute.

If the scheduled event time for the top process is equal to the current time the symbol >> will be placed to the left of that process indicating that this process is currently running.

The window's top status bar contains the current simulation time and is read: **Current Time:** <**time**>.

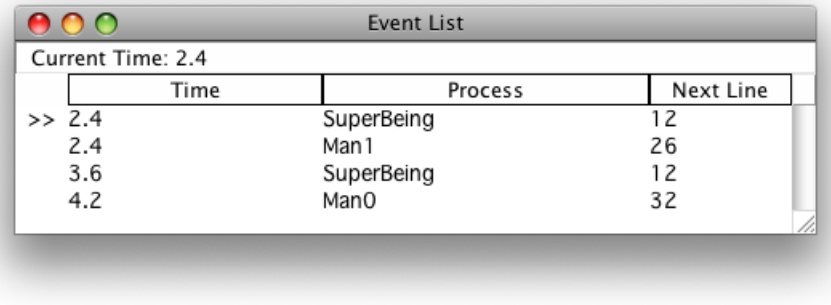

Figure 1: Sample Event List Window

4.2 GenericWindow Class

The GenericWindow Class is used to create objects that control generic windows. The windows print a user defined hook method. It is subclassed by both the ProcessWindow, and the ResourceWindow.

4.3 ProcessWindow

The ProcessWindow Class is used to create objects that control process windows. The process window prints out the current status of the process: *Active* or *Passive*, the next scheduled event simulation time, interrupted status, whether the process is currently running, and a user defined hook.

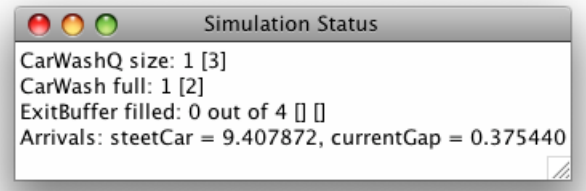

Figure 2: Sample Generic Window with hook

Figure 3: Sample Process Window

4.4 ResourceWindow Class

The ResourceWindow Class is used to create objects that control resource windows. Along with the user defined hook, the resource window contains two tables:

- **ActiveQ Table:** Lists the names of all processes that are currently actively using this resource.
- **WaitQ Table:** Lists the names of all processes that are currently queuing for this resource.

5 Example

The following code, **Example.py** gives an example as to how to use SimulationGUIDebug. Important lines are explained in more detail below.

```
1  # Example.py - Example for SimulationGUIDebug
2  from SimulationGUIDebug import * # import SimulationGUIDebug
3
4  # Creates man
5  class SuperBeing(Process):
6          def __init__(self,earth):
7                  Process.__init__(self)
8                  self.earth = earth
9
10         def Create(self):
11                 while True:
```

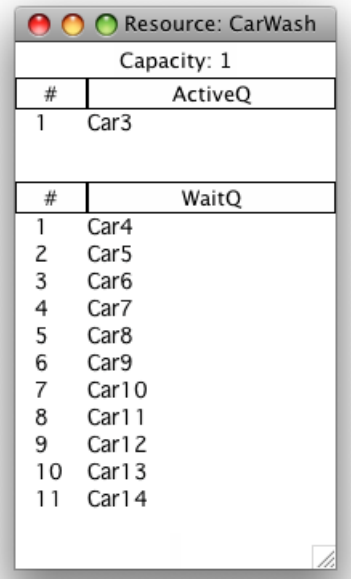

Figure 4: Sample Resource Window

```
                        yield hold,self,1.20
                        man = Man(self.earth)
                        activate(man,man.Walk())
                        register(man,man.Status) # register the man instance with hook

# Man waits for earth resource, then becomes, baby, adult, and leaves earth
class Man(Process):
        ID = 0
        def __init__(self,earth):
                Process.__init__(self,name="Man%d"%Man.ID) # set name to ensure window title is set
                self.earth = earth
                self.status = "in heaven"
                Man.ID += 1

        def Walk(self):
                self.status = "waiting for earth "
                yield  request,self,self.earth
                self.status = "baby"
                yield  hold,self,1
                self.status = "adult"
                yield  hold,self,2
                self.status = "good bye earth"
                yield  release,self,self.earth

        def Status(self):
                return self.status

# set up
initialize()
register(SuperBeing,name="SuperBeing") # register SuperBeing class with name

Earth = Resource(2,name="Earth") # set name to ensure window title is set
register(Earth) # register Earth Resource

SB = SuperBeing(Earth)
activate(SB,SB.Create()) # when activated SB will be registered
```

```
48
49  # simulate
50  simulate(until=1000)
```

- **Line 2:** Import *SimulationGUIDebug* here.
- **Lines 21,43:** Here the *name* variable is set so that the registered windows may use it as the title.
- **Line 15:** Here the instance *man* is registered with the user hook as *man.Status* and window is created for the process.
- **Line 41:** Here the SuperBeing class (NOT an instance) is registered. So now whenever a SuperBeing instance is activated SimulationGUIDebug will automatically register it using the *name* parameter S̈uperBeingäs the window's title.
- **Line 44:** Here the resource Earth is registered and a window is created for the resource.
- **Line 47:** Note that SB was not registered here. That is because its class was already registered on line 40. Once SB is activated, it will automatically be registered.

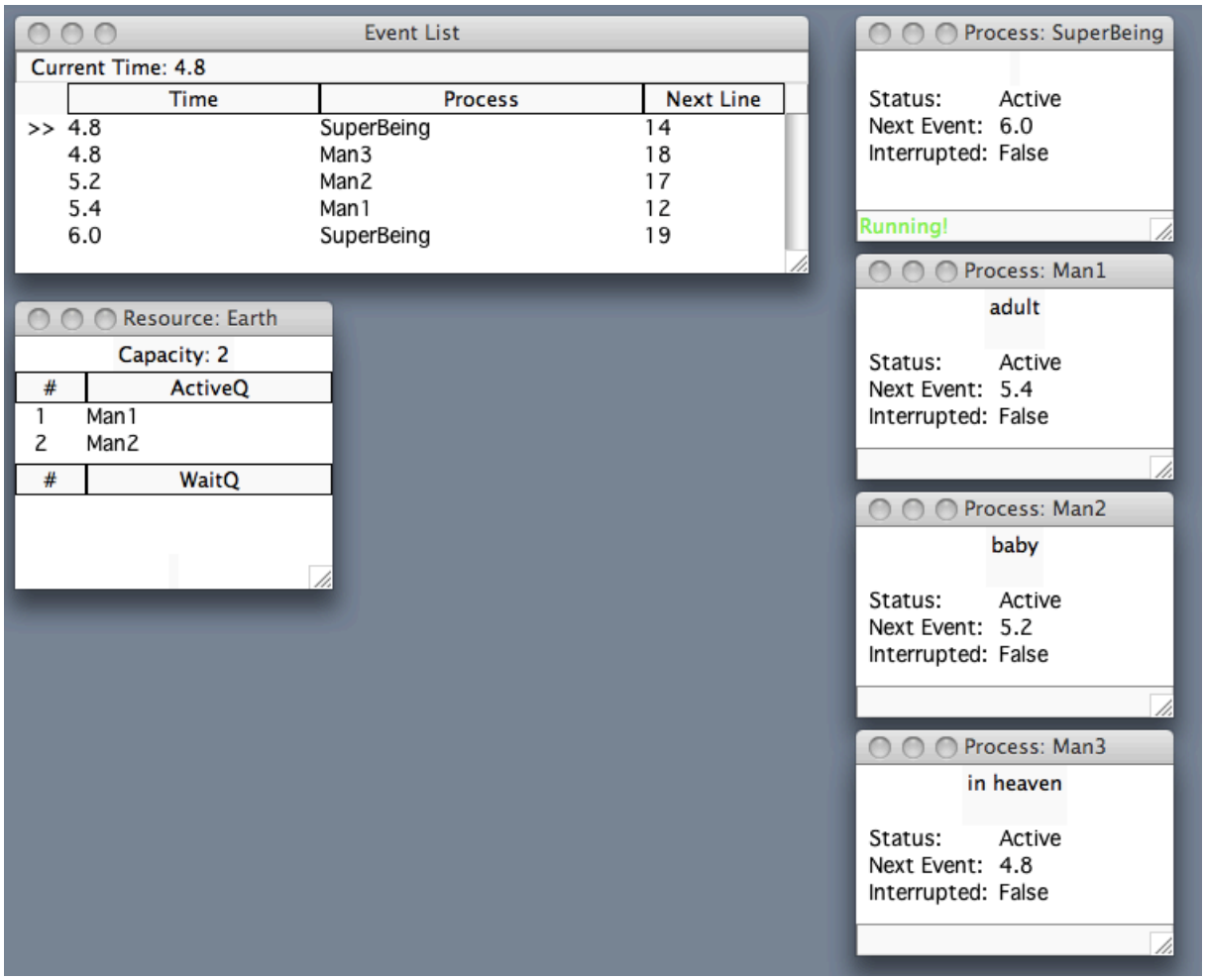

Figure 5: Example.py Preview

6 SimulationGUIDebug Backend

6.1 Backend Goal

When creating the Debugging backend, one of our primary goals was to make the interface for the user similar to other familiar debuggers and require as little change to the program as possible. In order to use the debugger, the program just needs to include SimulationGUIDebug, instead of SimPy, by using from SimulationGUIDebug import *. Otherwise, all function calls to our debugger are the same as they would be in SimPy. Also, the interactive command-line interface works very much like PDB or DDD does, with basic commands performing the debugging.

6.2 Backend Implementation

Our actual debugger works through SimPy's callback functionality. Instead of letting the user specify a callback function to SimPy, we use our own callbackFunction() function. Since SimPy calls this callback function after every event has fired, our callbackFunction() simply needs to see if there is a breakpoint specified for the event's time and, if so, perform our GUI updates. Since SimPy's callback functionality is blocking, so that the simulation is suspended while the callback function is executing, the simulation is truly stopping when a breakpoint is hit. Since we are using SimPy's callback function ourselves, we implement our own callback functionality to allow the user to still use this feature.

To get the interaction with the user, there is a function promptUser(), which uses Python's raw_input() function to get the commands from the user. The basic debugging functions exist: [s] to step to the next event in the simulation, [c] to continue to the next breakpoint, [b #] to add a breakpoint (tuples are allowed, in order to add multiple breakpoints), [q] to quit the simulation. We also save the last command that was issued, so entering no command will run the previously executed command, much like other debuggers.

In addition to our debugger, we wanted users to be able to utilize existing debuggers to work with their programs. In order to achieve this goal, we added a feature which allows the user to specify the run mode, via the function setRunMode(). The default value is STEP, which runs the debugger as expected. The alternate value is NO_STEP, which will not evaluate breakpoints at all, but continues to update the GUI. This way, a user can specify breakpoints and step through the program in an external debugger such as PDB, but still utilize our GUI for debugging purposes.

7 GUIDebug.py (GUI Implementation)

GUIDebug is the GUI package that SimulationGUIDebug uses to create its GUI. Users who wish to use SimulationGUIDebug do not to know the details of GUIDebug. The following information is useful for users who wish to expand the GUI capabilities of SimulationGUIDebug.

7.1 GUIController Class

7.1.1 Use

The GUIController is used to control all of the GUI windows that are currently open, and to create new windows. It currently contains methods to add a new GenericWindow, ProcessWindow, and ResourceWindow

as well as to update the GUI of all currently opened windows.

7.1.2 GUIController API

addNewWindow

- **Call:** *addNewWindow(self,obj,name,hook)*
- **Description:** Creates and adds a new GenericWindow object to the controller's window list.
- **Parameters:** *obj* is the object to be associated with the window. *hook* is a function that returns a string that will be printed to the window.
- **Return Value:** None

addNewProcess

- **Call:** *addNewProcess(self,obj,name,hook)*
- **Description:** Creates and adds a new ProcessWindow object to the controller's process window list.
- **Parameters:** *obj* is the object to be associated with the window. *hook* is a function that returns a string that will be printed to the window.
- **Return Value:** None

addNewResource

- **Call:** *addNewResource(self,obj,name,hook)*
- **Description:** Creats and adds a new ResourceWindow object to the controller's resource window list.
- **Parameters:** *obj* is the object to be associated with the window. *hook* is a function that returns a string that will be printed to the window.
- **Return Value:** None

updateAllWindows

- **Call:** *updateAllWindows(self)*
- **Description:** Calls update() for each window in the controller's window list. Then calls organizeWindows() to clean up and organize the current windows on the screen.
- **Parameters:** None
- **Return Value:** None

saveNextEvent

- **Call:** *saveNextEvent(self)*

- **Description:** Saves the next event that will be run in self.nextEvent.
- **Parameters:** None
- **Return Value:** None

removeWindow

- **Call:** *removeWindow(w)*
- **Description:** Removes all instances of w in GUIController's window lists.
- **Parameters:** *w*: GenericWindow to be removed from controller.
- **Return Value:** None

organizeWindows

- **Call:** *organizeWindows(self)*
- **Description:** Organizes the windows so that the event list window is in the top left corner of the screen, the process windows are to the right of the event list window, and the resource windows are below the event list window.
- **Parameters:** None
- **Return Value:** None

8 Source Code

8.1 SimulationGUIDebug.py

```
1  from SimPy.SimulationStep import *
2  from Tkinter import *
3  import SimPy.SimulationStep, GUIDebug
4
5  # global variables
6  _breakpoints = []
7  _until = 0
8  _callback = None
9  _lastCommandIssued = ""
10 _simStarted = False
11 _registeredClasses = []
12 _runMode = None
13
14 # run modes
15 STEP = 1
16 NO_STEP = 2
17
18 # register new object for windowing
19 def register(obj,hook=lambda :"",name=None):
20
21         global _registeredClasses
22
23         # if process subclass is given register it
24         if type(obj) == TypeType and issubclass(obj, Process):
25                 _registeredClasses += [(obj,name,hook)]
```

```
26
27          # if instance of process is given register it
28          elif issubclass(type(obj), Process):
29                  _guiCtrl.addNewProcess(obj,name,hook)
30
31          # if instance of Resource is given register it
32          elif issubclass(type(obj), Resource):
33                  _guiCtrl.addNewResource(obj,name,hook)
34
35          # else create a generic window with hook
36          else:
37                  _guiCtrl.addNewWindow(obj,name,hook)
38
39  # override activate to catch registered class instances
40  def activate(obj,process,at="undefined",delay="undefined",prior=False):
41
42          global _registeredClasses
43
44          SimPy.SimulationStep.activate(obj,process,at,delay,prior)
45
46          # if obj is instance of the class register it
47          for c,n,h in _registeredClasses:
48                  if isinstance(obj, c):
49                          _guiCtrl.addNewProcess(obj,n,h)
50
51  # add to breakpoints
52  def newBreakpoint(newBpt):
53
54          global _breakpoints
55          _breakpoints.append(newBpt)
56          _breakpoints.sort()
57
58  # set the current run mode of simulation
59  def setRunMode(runMode):
60
61          global _runMode
62          _runMode = runMode
63
64  # initialize the simulation and the GUI
65  def initialize():
66
67          SimPy.SimulationStep.initialize()
68
69          # create gui controller
70          global _guiCtrl
71          _guiCtrl = GUIDebug.GUIController()
72
73          # initialize run mode if not already set
74          global _runMode
75          if not _runMode:
76                  _runMode = STEP
77
78  # simulation function
79  def simulate(callback=lambda :None, until=0):
80
81          global _runMode
82
83          # print usage
84          if( _runMode == STEP ):
85                  print "Breakpoint Usage:"
86                  print "  [c]   Continue simulation"
87                  print "  [s]   Step to next event"
88                  print "  [b #] Add new breakpoint"
89                  print
90                  print "  [q]   Quit debugger"
91                  print
92
93          # set global variables
```

```
94          global _until
95          _until = until
96
97          global _callback
98          _callback = callback
99
100         # initialize to step command
101         global _lastCommandIssued
102         _lastCommandIssued = "s"
103
104         # only prompt user if we are in STEP mode
105         if( _runMode == STEP): promptUser()
106
107         # quit if user entered 'q'
108         if( _lastCommandIssued == 'q'):
109                 return
110
111         # begin simulation
112         global _simStarted
113         _simStarted = True
114         startStepping()
115         SimPy.SimulationStep.simulate(callback=callbackFunction,until=_until)
116
117 # check for breakpoints
118 def callbackFunction():
119
120         global _breakpoints,_runMode,_guiCtrl
121
122         # NO_STEP mode means we update windows and take no breaks
123         # this is used for compatibility with REAL debuggers
124         if( _runMode == NO_STEP ):
125                 _guiCtrl.updateAllWindows()
126                 return
127
128         if( 0 == len(_breakpoints) ):
129                 return
130
131         # this is a breakpoint
132         if( now() >= _breakpoints[0] ):
133
134                 # update gui
135                 _guiCtrl.updateAllWindows()
136
137                 # remove past times from breakpoints list
138                 while( 0 != len(_breakpoints) and now() >= _breakpoints[0] ):
139                         _breakpoints.pop(0)
140
141                 # call user's callback function
142                 global _callback
143                 _callback()
144
145                 promptUser()
146
147 # prompt user for next command
148 def promptUser():
149
150         global _simStarted
151
152         # set prompt text
153         prompt = '(SimDB) > '
154
155     # pause for breakpoint
156         while( 1 ):
157                 input = raw_input( prompt )
158
159                 # take a look at the last command issued
160                 global _lastCommandIssued
161
```

```
162                 if 0 == len(input):
163                         input = _lastCommandIssued
164 
165                 _lastCommandIssued = input
166 
167                 # continue
168                 if( "c" == input ):
169                         break
170 
171                 # step
172                 elif( "s" == input ):
173                         global _breakpoints
174                         _breakpoints.insert(0,0)
175                         break
176 
177                 # add breakpoint
178                 elif( 0 == input.find("b")):
179                         try:
180                                 for i in eval( input[1:] + "," ):
181                                         newBreakpoint( int(i) )
182                         except SyntaxError:
183                                 print "missing breakpoint values"
184 
185                 # quit
186                 elif( "q" == input ):
187                         SimPy.SimulationStep.stopSimulation()
188                         return
189 
190                 else:
191                         print "  unknown command"
```

8.2 GUIDebug.py

```
1  from Tkinter import *
2  from SimPy.SimulationStep import now,Globals
3  
4  # Creates and controls the GUI of the program
5  class GUIController(object):
6  
7          def __init__(self):
8                  self.root = Tk()
9                  self.root.withdraw()
10 
11                 self.saveNextEvent()
12 
13                 self.eventWin = EventWindow(self)
14                 self.wlist = []
15                 self.plist = []
16                 self.rlist = []
17 
18         # Adds a new Window to the GUI
19         def addNewWindow(self,obj,name,hook):
20                 self.wlist += [GenericWindow(obj,hook,self,name)]
21 
22         # Adds a new Process to the GUI
23         def addNewProcess(self,obj,name,hook):
24                 self.plist += [ProcessWindow(obj,hook,self,name)]
25 
26         # Adds a new Resource to the GUI
27         def addNewResource(self,obj,name,hook):
28                 self.rlist += [ResourceWindow(obj,hook,self,name)]
29 
30         # Updates all the windows currently up
31         def updateAllWindows(self):
32 
33                 for w in self.wlist: w.update()
```

```
34                  for p in self.plist: p.update()
35                  for r in self.rlist: r.update()
36                  if self.eventWin.window: self.eventWin.update()
37
38                  self.organizeWindows()
39
40                  self.saveNextEvent()
41
42          # removes all instances of window in lists
43          def removeWindow(self, w):
44                  f = lambda win: win is not w
45                  self.wlist = filter(f,self.wlist)
46                  self.plist = filter(f,self.plist)
47                  self.rlist = filter(f,self.rlist)
48
49          # save next event to be run
50          def saveNextEvent(self):
51
52
53                  tempList=[]
54                  tempList[:]=Globals.sim.e.timestamps
55                  tempList.sort()
56
57                  for ev in tempList:
58
59                          # return only event notices which are not cancelled
60                          if ev[3]: continue
61
62                          # save next event
63                          self.nextEvent = ev
64                          return
65
66                  self.nextEvent = (None,None,None,None)
67
68          def organizeWindows(self):
69
70                  # event window
71                  eventWindowHeight = 0
72
73                  # only organize event window only if it exists
74                  if self.eventWin.window:
75                          eventWindowHeight = 40 + 20 * self.eventWin.table.size()
76                          self.eventWin.setWindowSize(500, eventWindowHeight ,20,40)
77
78                  # generic windows
79                  count = -1
80
81                  for win in self.wlist:
82                          count += 1
83
84                          (w,h,x,y) = win.getWindowSize()
85                          win.setWindowSize(w, h, 20, 40 + eventWindowHeight + 40 )
86
87                          eventWindowHeight += h + 40
88
89                  # process windows
90                  xCount = -1
91                  yCount = 0
92
93                  for p in self.plist:
94                          xCount += 1
95
96                          yCoord = 40 + 150 * xCount
97                          xCoord = 550 + 210 * yCount
98
99                          p.setWindowSize(200,120, xCoord, yCoord )
100
101                         if yCoord >= 600:
```

```
102                                 xCount = -1
103                                 yCount += 1
104
105                 # resource windows
106                 count = -1
107                 for r in self.rlist:
108                         count += 1
109
110                         windowHeight = 0
111                         windowHeight += 20 # capacity title
112                         windowHeight += 105 # empty table sizes
113
114                         windowHeight += (r.activeT.size() + r.waitT.size()) * 17 # add size for each row
115
116                         r.setWindowSize(200, windowHeight , 20 + 220 * count , 40 + eventWindowHeight + 40)
117
118
119
120 # Creates a basic window that shows a user made hook.
121 class GenericWindow(object):
122
123         def __init__(self, obj, hook, guiCtrl, title=None):
124                 self.window = Toplevel()
125                 self.window.protocol("WM_DELETE_WINDOW", self._destroyWindow)
126                 self.obj = obj
127                 self.hook = hook
128                 self.guiCtrl = guiCtrl
129                 if not title:
130                         self.title = "%s%s" % (type(obj),id(obj))
131                 else:
132                         self.title = title
133                 self.initGUI()
134
135
136         def setWindowSize(self,w,h,x,y):
137                 newG = "%dx%d+%d+%d" % (w,h,x,y)
138                 self.window.geometry(newG)
139
140         def setWindowOrigin(self,x,y):
141                 (w,h,xx,yy) = self.getWindowSize()
142                 newG = "%dx%d+%d+%d" % (w,h,x,y)
143                 self.window.geometry(newG)
144
145         def getWindowSize(self):
146                 g = self.window.geometry()
147                 return [int(i) for i in g.replace('+','x').split('x')]
148
149         def _destroyWindow(self):
150                 self.window.destroy()
151                 self.window = None
152                 self.guiCtrl.removeWindow(self)
153
154         # Creates the window
155         def initGUI(self):
156                 self.window.title(self.title)
157                 txt = self.hook()
158                 if txt != "": txt += '\n'
159                 self.hookTxt = Label(self.window,text=txt,justify=LEFT)
160                 self.hookTxt.pack()
161
162         # Updates the window
163         def update(self):
164                 txt = self.hook()
165                 if txt != "": txt += '\n'
166                 self.hookTxt["text"] = txt
167
168 # Class that creates the event window for the simulation that
169 # displays the time and event.
```

```
170 class EventWindow(GenericWindow):
171
172         def __init__(self, guiCtrl):
173                 self.window = Toplevel()
174                 self.window.protocol("WM_DELETE_WINDOW", self._destroyWindow)
175                 self.guiCtrl = guiCtrl
176                 self.initGUI()
177
178         # Creates the initial window using a two column window with a
179         # status bar on the bottom
180         def initGUI(self):
181                 self.window.title("Event List")
182                 # Creates the table
183                 self.table = MultiListbox(self.window,((’’, 1), (’Time’,15),
184                                                 (’Process’,20),(’Next Line’,5)))
185                 # Adds the status bar to display the current simulation time
186                 self.status = StatusBar(self.window)
187                 self.status.pack(side=TOP, fill=X)
188
189                 self.update()
190
191         # Updates the window
192         def update(self):
193                 self.updateETable()
194                 self.updateStatus()
195
196         # Updates the status bar
197         def updateStatus(self):
198                 self.status.set("  Current Time: %s",now())
199
200         # Updates the table
201         def updateETable(self):
202
203                 self.table.delete(0,self.table.size())
204
205                 tempList=[]
206
207                 tempList.sort()
208                 tempList[:] = Globals.sim._e.timestamps
209                 ev = self.guiCtrl.nextEvent
210
211                 nextLine = 0
212                 if( ev[2] ):
213                         if( ev[2]._nextpoint ):
214                                 nextLine = ev[2]._nextpoint.gi_frame.f_lineno
215
216                 if ev[0]:
217                         self.table.insert(END,(’  >>’,
218                            str(ev[0]), ev[2].name, nextLine ))
219
220                 count = -1
221                 for ev in tempList:
222
223                         # return only event notices which are not cancelled
224                         if ev[3]: continue
225
226                         count += 1
227
228                         currentEvent = ’’
229                         #if count == 0 and now() == ev[0]:
230                         #        currentEvent = ’  >>’
231
232                         nextLine = 0
233                         if( ev[2] ):
234                                 if( ev[2]._nextpoint ):
235                                         nextLine = ev[2]._nextpoint.gi_frame.f_lineno
236
237                         self.table.insert(END,(currentEvent,
```

```
238                                     str(ev[0]), ev[2].name, nextLine ))
239
240                 self.table.pack(expand=YES,fill=BOTH)
241
242 # Creates a Process Window that shows the status, Next Event time,
243 # if the Process is currently interupted, and an optional user hook.
244 class ProcessWindow(GenericWindow):
245
246         def __init__(self, obj, hook, guiCtrl, name):
247                 self.proc = obj
248                 if name:
249                         obj.name = name
250                 GenericWindow.__init__(self, obj, hook, guiCtrl, "Process: %s" % obj.name)
251
252         # Initializes the window
253         def initGUI(self):
254
255                 # Creates the table
256                 self.table = MultiListbox(self.window,((None,10),(None,15)))
257                 self.status = StatusBar(self.window)
258                 self.status.pack(side=BOTTOM, fill=X)
259
260                 GenericWindow.initGUI(self)
261                 self.setWindowSize(0,0,-1000,-1000)
262
263                 self.update()
264
265         # Updates the window
266         def update(self):
267
268                 # If the process has been terminated close the window
269                 if self.proc.terminated():
270                         self._destroyWindow()
271                         return
272
273                    if self.isRunning():
274                         self.status.label["text"] = "Running!"
275                            self.status.label["fg"] = "green"
276                    else:
277                         self.status.label["text"] = ""
278                            self.status.label["fg"] = "white"
279
280                 self.table.delete(0,self.table.size())
281
282                 if self.proc.active() == False:
283                         status = "Passive"
284                 else:
285                         status = "Active"
286
287                 if self.proc._nextTime:
288                         nextEvent = self.proc._nextTime
289                 else:
290                         nextEvent = ""
291
292                 if self.proc.interrupted() == True:
293                         interrupted = "True"
294                 else:
295                         interrupted = "False"
296
297                 self.table.insert(END,("  Status:", status))
298                 self.table.insert(END,("  Next Event:", nextEvent))
299                 self.table.insert(END,("  Interrupted:", interrupted ))
300
301                 self.table.pack(expand=YES,fill=BOTH)
302
303                 GenericWindow.update(self)
304
305         def isRunning(self):
```

```
306                 return self.guiCtrl.nextEvent[2] is self.proc
307
308 # Creates a Resource Window that displays the capacity, waitQ,
309 # activeQ and an optional user hook
310 class ResourceWindow(GenericWindow):
311
312         def __init__(self,obj,hook,guiCtrl,name):
313                 self.resource = obj
314                 if name:
315                         obj.name = name
316                 GenericWindow.__init__(self,obj,hook, guiCtrl,"Resource: %s" % obj.name)
317
318         # Initializes the window with the two tables for the waitQ and activeQ
319         def initGUI(self):
320                 Label(self.window,text="Capacity: %d" % self.resource.capacity).pack()
321
322                 self.activeT = MultiListbox(self.window,(('#',5),('ActiveQ',20)))
323                 self.waitT = MultiListbox(self.window,(('#',5),('WaitQ',20)))
324                 self.updateQTables()
325
326                 GenericWindow.initGUI(self)
327                 self.setWindowSize(0,0,-1000,-1000)
328
329         # Updates the window
330         def update(self):
331                 GenericWindow.update(self)
332                 self.updateQTables()
333
334         # Updates the waitQ and activeQ tables
335         def updateQTables(self):
336                 self.activeT.delete(0,END)
337                 self.waitT.delete(0,END)
338                 # Update the activeQ
339                 for i in range(len(self.resource.activeQ)):
340                         col1 = '%d' % (i+1)
341                         col2 = self.resource.activeQ[i].name
342                         self.activeT.insert(END,("   " + col1,col2))
343                 # Update the waitQ
344                 for i in range(len(self.resource.waitQ)):
345                         col1 = '%d' % (i+1)
346                         col2 = self.resource.waitQ[i].name
347                         self.waitT.insert(END,("   " + col1,col2))
348
349                 self.activeT.pack(expand=YES,fill=BOTH)
350                 self.waitT.pack(expand=YES,fill=BOTH)
351                 self.window.update()
352
353 # A class that creates a multilistbox with a scrollbar
354 class MultiListbox(Frame):
355         def __init__(self, master, lists):
356                 Frame.__init__(self, master)
357                 self.lists = []
358                 for l,w in lists:
359                         frame = Frame(self); frame.pack(side=LEFT, expand=YES, fill=BOTH)
360
361                         if l is None:
362                                 None
363                         elif l is '':
364                                 Label(frame, text='', borderwidth=1, relief=FLAT).pack(fill=X)
365                         else:
366                                 Label(frame, text=l, borderwidth=1, relief=SOLID).pack(fill=X)
367
368                         lb = Listbox(frame, width=w, height=0, borderwidth=0, selectborderwidth=0,
369                                  relief=FLAT, exportselection=FALSE)
370                         lb.pack(expand=YES, fill=BOTH)
371                         self.lists.append(lb)
372                 frame = Frame(self); frame.pack(side=LEFT, fill=Y)
373                 Label(frame, borderwidth=1, relief=RAISED).pack(fill=X)
```

```
374                 sb = Scrollbar(frame, orient=VERTICAL, command=self._scroll)
375                 sb.pack(expand=YES, fill=Y)
376                 self.lists[0]['yscrollcommand']=sb.set
377
378         def _select(self, y):
379                 row = self.lists[0].nearest(y)
380                 self.selection_clear(0, END)
381                 self.selection_set(row)
382                 return 'break'
383
384         def _button2(self, x, y):
385                 for l in self.lists: l.scan_mark(x, y)
386                 return 'break'
387
388         def _b2motion(self, x, y):
389                 for l in self.lists: l.scan_dragto(x, y)
390                 return 'break'
391
392         def _scroll(self, *args):
393                 for l in self.lists:
394                         apply(l.yview, args)
395
396         def curselection(self):
397                 return self.lists[0].curselection()
398
399         def delete(self, first, last=None):
400                 for l in self.lists:
401                         l.delete(first, last)
402
403         def get(self, first, last=None):
404                 result = []
405                 for l in self.lists:
406                         result.append(l.get(first,last))
407                 if last: return apply(map, [None] + result)
408                 return result
409
410         def index(self, index):
411                 self.lists[0].index(index)
412
413         def insert(self, index, *elements):
414                 for e in elements:
415                         i = 0
416                         for l in self.lists:
417                                 l.insert(index, e[i])
418                                 i = i + 1
419
420         def size(self):
421                 return self.lists[0].size()
422
423         def see(self, index):
424                 for l in self.lists:
425                         l.see(index)
426
427         def selection_anchor(self, index):
428                 for l in self.lists:
429                         l.selection_anchor(index)
430
431         def selection_clear(self, first, last=None):
432                 for l in self.lists:
433                         l.selection_clear(first, last)
434
435         def selection_includes(self, index):
436                 return self.lists[0].selection_includes(index)
437
438         def selection_set(self, first, last=None):
439                 for l in self.lists:
440                         l.selection_set(first, last)
441
```

```
442 # Creates a statusbar
443 class StatusBar(Frame):
444
445     def __init__(self, master):
446         Frame.__init__(self, master)
447         self.label = Label(self, bd=1, relief=SUNKEN, anchor=W)
448         self.label.pack(fill=X)
449
450     def set(self, format, *args):
451         self.label.config(text=format % args)
452         self.label.update_idletasks()
453
454     def clear(self):
455         self.label.config(text="")
456         self.label.update_idletasks()
```