Miss Grant's Controller: A JSD specification

by Stephen Ferg
http://www.ferg.org
revised: 2010-11-20

For the purpose of writing computer software specifications, it is useful to view a computer software system as a software "machine" that transitions from state to state under the control of an input stream of events.

Traditionally, computer system specifications focus on the states and the transitions between the states: they view the system as a state machine and use state transition diagrams (STDs) to specify the behavior of the machine.

In contrast, Jackson System Development (JSD)[1] specifications focus on the events and the sequence in which the events may occur. JSD views a software machine as a simulation in which model processes (coroutines running inside the system) simulate real-world processes. The model processes running in the machine are synchronized with their real-world counterparts by means of events – events sent from the real-world processes into the machine. JSD uses *action structure diagrams* to represent model processes.

I (and others, of course) believe that JSD-style specifications are a more useful tool than STDs and state machines for specifying many systems. For many clients, events and the order in which events occur are "easier to think with" than states and state transitions. [2]

In this paper, I will present a small argument-by-example for JSD event-oriented specifications. I will present the example, and you can decide for yourself what you think of it.

---

[1] See www.jacksonworkbench.co.uk/stevefergspages/jackson_methods

[2] Although many clients find event-oriented **concepts** easier to use, they typically find state transition **diagrams** easier. Flow charts -- Dilbert's "circles and arrows" -- have been used in business documentation for decades. State transition diagrams seem familiar and intuitive because they look and work like flow charts. Ease-of-use in this case is a result of familiarity rather than superiority.

The problem and its state-oriented specification

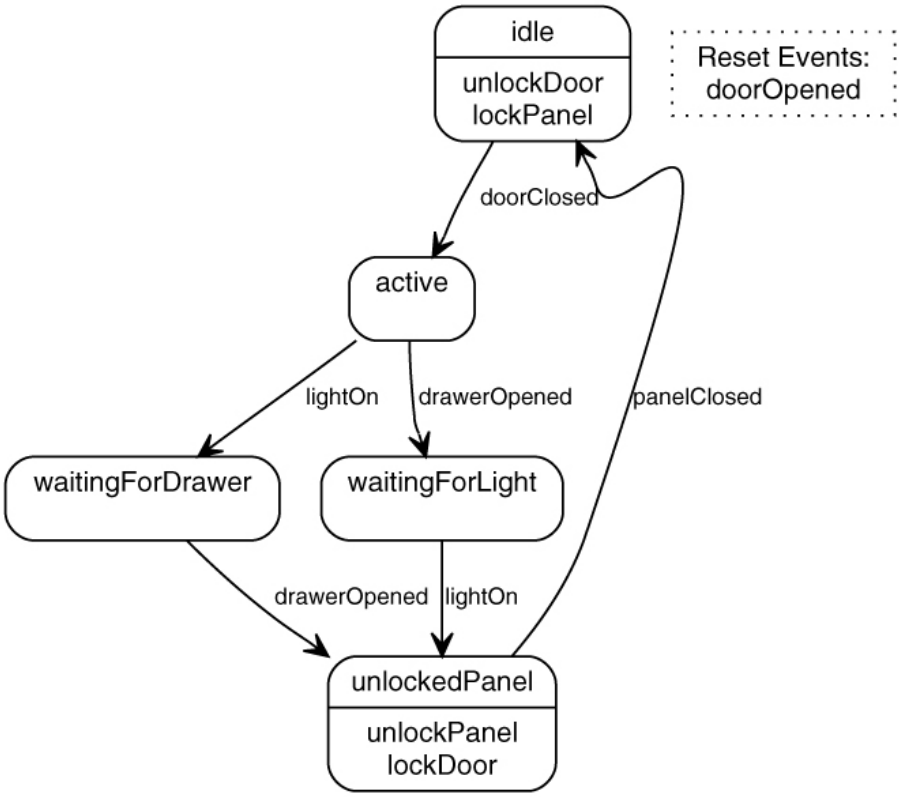
My example problem will be based on an example problem from the introduction to Martin Fowler's new (2010) book *Domain-Specific Languages*.

> http://www.amazon.com/Domain-Specific-Languages-Addison-Wesley-Signature-Martin/dp/0321712943/ref=sr_1_1?ie=UTF8&s=books&qid=1289683183&sr=1-1

I've chosen it because Fowler has done an excellent job of presenting a state-oriented approach to the problem, and he has made his work available on the Web as *Domain-Specific Languages: An Introductory Example*. See http://www.informit.com/articles/printerfriendly.aspx?p=1592379

I have compressed and partly re-written Fowler's original description of the problem. For Fowler's original statement of the problem, see the Web link listed above.

> I have childhood memories of watching cheesy adventure films on TV. Often, these films would be set in some old castle and feature secret compartments or passages. In order to find them, heroes would need to pull the candle holder at the top of stairs and tap the wall twice.
>
> Let's imagine a company, Gothic Security Systems, that decides to build security systems based on this idea.
>
> At the center of their security systems is some controller software that listens to event messages, figures out what to do, and sends command messages to devices (like the locks of panels that hide secret compartments).
>
> A good way to think about the controller is as a state machine. Each sensor sends an event that can change the state of the controller. As the controller enters a state, it can send a command message out to devices on the network.
>
> Each customer who buys a gothic security system can have it configured to his or her individual needs.
>
> Take Miss Grant, for example. Miss Grant has a secret compartment in her bedroom that is normally locked and concealed. In order to access her secret compartment, Miss Grant must: (1) close her bedroom door, and then (in either order) (2a) open the second drawer in her chest and (2b) turn on her bedside light.
>
> Once these are done, the secret panel is unlocked for her to open.
>
> I can represent this sequence as a state diagram.



On the STD, some states have notations indicating the command messages that the controller sends to the devices when it enters the state.

- The *unlockedPanel* state has notations for *unlockPanel* and *lockDoor* commands. If the controller is in the *waitingForLight* state, and Miss Grant turns on the light, then the controller sends the *unlockPanel* message to the appropriate network device (the lock on the panel that conceals the secret compartment), sends the *lockDoor* command to the lock on the bedroom door, and then transitions itself to the *unlockedPanel* state.

- If the controller is in the *unlockedPanel* state, and Miss Grant either closes the panel hiding the secret compartment, or opens the bedroom door, then the controller sends the *unlockDoor* and *lockPanel* commands and transitions itself to the *idle* state.

There are two ways in which Fowler's state transition diagram differs from a conventional STD.

First, as Fowler notes:

> The controller is, mostly, a simple and conventional state machine, but there is a twist.
>
> The controller spends most of its time in the idle state. Certain events ("reset events") can jump the controller back into this idle state from any of the other states, effectively resetting the model. In Miss Grant's case, opening the door is such a reset event.
>
> On the STD, the existence of reset events is indicated by the dotted box near the idle state. Classic STDs don't include reset events. Introducing reset events adds a twist that is unique to this context.
>
> Note that reset events aren't necessary to express Miss Grant's controller. The alternative would be for every state to have a transition, triggered by the *doorOpened* event, to the idle state. Lines for all of those transitions would clutter up the diagram and reduce its readability. The notion of a reset event is useful because it simplifies the diagram by allowing us to avoid showing all of those reset-related transitions.

The second way in which Fowler's state transition diagram differs from a conventional STD is that it doesn't show all transitions, it shows only transitions that change state.

Suppose that Miss Grant comes into the room without closing the door behind her, leaving the controller in the *idle* state. She turns on the light. The *lightOn* event is an event that the controller recognizes, although a *lightOn* event has no particular meaning to the controller when it is in the *idle* state.

In a conventional STD, the controller's handling of the *lightOn* event would be shown by a circular transition out of the *idle* state and then back into the *idle* state. The result would be that the controller does actually process the *lightOn* event when it is in the *idle* state, but the processing doesn't change the controller's state.

These circular transitions are not shown on Fowler's STD.

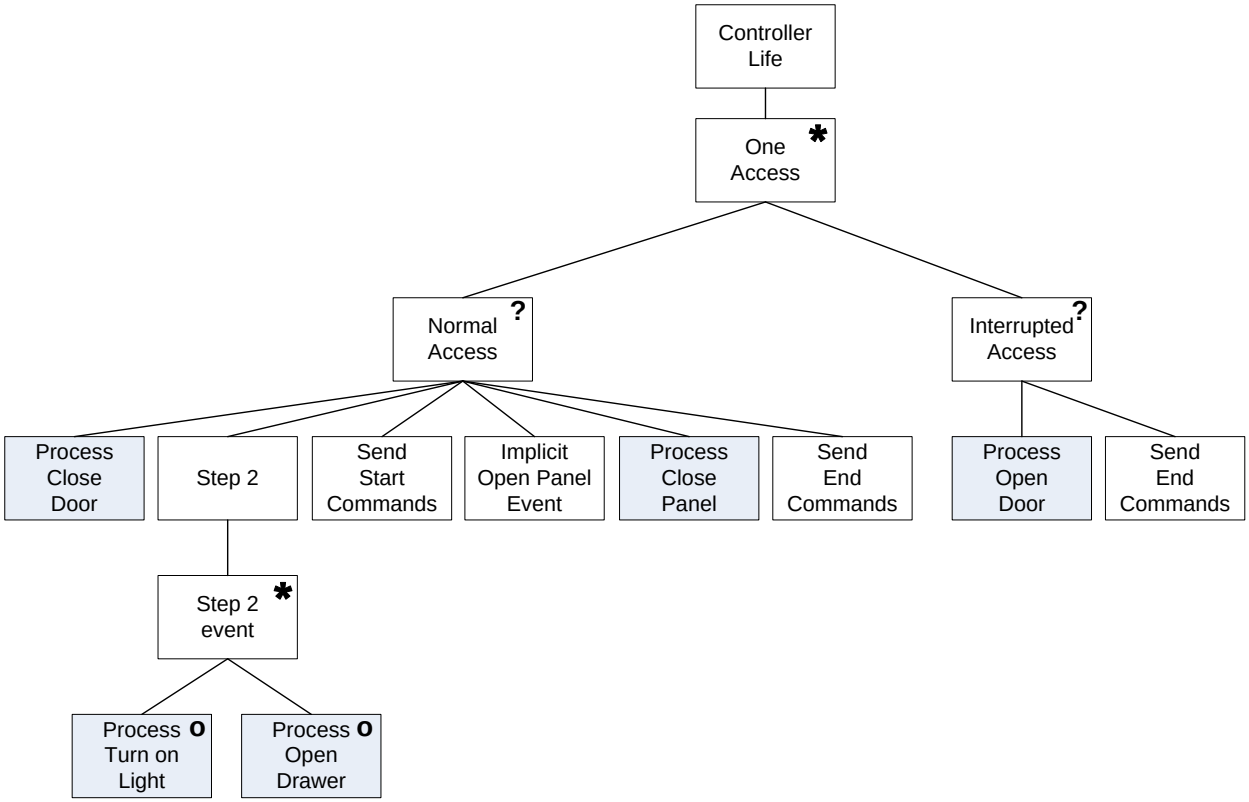
The problem and its event-oriented specification

State-oriented specifications use *state-transition diagrams* to show states and transitions.

JSD, an event-oriented method, uses *action structure diagrams* to show events and the sequences in which events can occur.

Here is a JSD action-structure diagram for Miss Grant's controller. The shaded boxes indicate the processing of an event. They correspond to the transitions on an STD.

Following Fowler's example, I've omitted showing the processing of reset events and the processing of events that don't change the controller's state.



What this diagram says is that the life of Miss Grant's controller consists of a sequence of indefinitely many *access* episodes.

A typical *normal access* begins with Miss Grant closing her bedroom door and then performing Step 2: a series of actions that must include both *turnOnLight* and *openDrawer* at least once.

At that point the controller issues the *startCommands* (the commands that really start the access episode: *lockDoor*, *unlockPanel*). Now, with the panel unlocked, Miss Grant opens the panel and presumably does something with the contents of the secret compartment.

When she has finished, she closes the panel. When she closes the panel, the controller issues the *endCommands* (the commands that really end the access episode: *lockPanel*, *unlockDoor*).

At that point the *normal access* is finished. In state-oriented terms: the controller returns to its *idle* state.

In most cases JSD isn't really interested in *states*. For JSD, a state is simply what a model process looks like between events. But if you want to think in terms of states, you can. Mapping the JSD model back to the STD, we can say that during a *normal access* episode:

after this event is processed	the state of the controller is
closeDoor	*active*
turnOnLight	waiting for both *turnOnLight* and *openDrawer* to have happened at least once
openDrawer	
after both *turnOnLight* and *openDrawer* have happened, and the *startCommands* have been issued	*unlockedPanel*
closePanel	*idle*

So that's what happens during a *normal access* episode.

Sometimes, while an access episode is in progress, an *openDoor* event occurs. The *openDoor* event interrupts the normal progress of the access episode. The episode is no longer a *normal access*; it becomes an *interrupted access*. The controller processes the *openDoor* event, issues the *end commands* (*lockPanel*, *unlockDoor*), and the *access* episode is over.

In JSD, the transition or jump from a *normal access* to an *interrupted access* is called a "quit". In our example, a JSD analyst would say that the *openDoor* event triggered a quit.

For all practical purposes, triggering a quit is the same as raising (or throwing) an exception. If we were writing a Java program to implement this model process, the code for the *normal access* would be placed in a "try" block. The code for the *interrupted access* would be placed in the corresponding "catch" block. And the code for triggering the quit might look something like this:

```
if (event.typeIs(OpenDoor)) {
      throw new JsdQuitException();
}
```

Executable specifications

For a long time, JSD experts have had a vision of *executable specifications:* the ability to write software specifications in a high-level pseudo-code that could be executed to test the specifications.[3]

In the 1990's that vision was frustrated primarily by the fact that a JSD model process is a coroutine,[4] but COBOL – the programming language in use in the business community where JSD was most popular – did not support coroutines. Executable specifications could be built in COBOL, but only by using an elaborate mechanism employing code generation, a macro-processor, many custom macros, and a lot of GOTOs. I know; I wrote a lot of it.

That situation has changed in the last few years, with the increasing acceptance of dynamic programming languages, and in particular the increasing use of Python. Python, it turns out, is the ideal language for creating executable JSD specifications.

- First of all, Python has an exceptionally clean and simple syntax – it has a well-deserved reputation as "executable pseudo-code". This makes it very easy to learn and to use.

- Second, Python is a dynamically-typed language with powerful introspection capabilities. This makes it extremely flexible and powerful.

- Finally, and most importantly, (since version 2.5) Python has native support for coroutines.[5]

Recently, I have been developing a small (currently, 176 lines) package called pyJSD. pyJSD adds some JSD syntactic sugar to Python, in order to make Python even more useful to JSD systems analysts.

Using Python and pyJSD, I have developed an executable specification of Miss Grant's controller.

This is what Miss Grant's controller looks like when the action structure diagram is translated into Python.[6]

---

[3] See "Executable Specifications as a Tool for System Architecture" by A. T. McNeile and J. Powell, in *JSP & JSD: The Jackson Approach to Software Development*, second edition, by John R. Cameron, IEEE Computer Society Press, 1989.

[4] "Coroutines are functions or procedures that save control state between calls." http://www.c2.com/cgi/wiki?CoRoutine accessed November 16, 2010.

"A coroutine is a routine that can be suspended at some point and resume from that point when control returns." http://gcu.googlecode.com/files/coroutine.pdf

See also http://en.wikipedia.org/wiki/Coroutine

[5] It's impossible to mention Python and coroutines in the same breath without mentioning the pioneering work of David Beazley. He proved that all of the capabilities needed to do executable specifications – even a trampolining scheduler – can be implemented in Python (see his PyCon 2009 presentation, *A Curious Course on Coroutines and Concurrency* at www.dabeaz.com/coroutines). Now, the only remaining question is whether the underlying machinery can be wrapped in enough syntactic sugar to make it a practical tool for working JSD analysts.

[6] The code is written in Python 3.0.

The opening lines set up the environment.

They define the kinds of events that the controller can receive from the sensors, and they define the devices to which commands can be sent.

```
001 #missGrantsController.py
002 import jsd
003
004 #Define sensor events that we can receive
005 class CloseDoor  (jsd.Event): pass
006 class TurnOnLight(jsd.Event): pass
007 class OpenDrawer (jsd.Event): pass
008 class ClosePanel (jsd.Event): pass
009 class OpenDoor   (jsd.Event): pass
010
011 #Define devices where we can send commands
012 class BedroomDoor:
013     def lock(self):
014         print("   door  locked")
015     def unlock(self):
016         print("   door  unlocked")
017
018 class CompartmentPanel:
019     def lock(self):
020         print("   panel locked")
021     def unlock(self):
022         print("   panel unlocked")
023
024 door  = BedroomDoor()
025 panel = CompartmentPanel()
 026
027 def sendStartCommands():
028     door.lock(); panel.unlock()
029
030 def sendEndCommands():
031     panel.lock(); door.unlock()
032
```

Here is the code for the model process, the controller itself.

```
033 #  Define the controller model process   ---------------
034 class MissGrantsController(jsd.ModelProcess):
035     def Life(self):
036         event = yield()                                    # JSD getFirstEvent
037         while True:                                        # do forever
038             # One Access
039             try:
040                 #-------------------------------------------------------------
041                 # JSD "POSIT" that we have a "normal access"
042                 #-------------------------------------------------------------
043                 # process event: closeDoor
044                 while not event.typeIs(OpenDoor, CloseDoor):
045                     event = yield()         # ignore irrelevant events
046                 if event.typeIs(OpenDoor): jsd.quit()      # the RESET event
047                 event = yield()                            # JSD getNextEvent
048
049                 # Step 2
050                 LightIsOn    = False
051                 DrawerIsOpen = False
052                 while True:              # loop until we BREAK out of the loop
053                     while not event.typeIs(OpenDoor, TurnOnLight, OpenDrawer):
054                         event = yield()     # ignore irrelevant events
055                     if event.typeIs(OpenDoor): jsd.quit()# the RESET event
056
057                     if event.typeIs(TurnOnLight):
058                         LightIsOn = True
059                     elif event.typeIs(OpenDrawer):
060                         DrawerIsOpen = True
061
062                     if LightIsOn and DrawerIsOpen:
063                         break
064                     else:
065                         event = yield()                    # JSD getNextEvent
066
067                 sendStartCommands()                        # to devices
068                 event = yield()                            # JSD getNextEvent
069
070                 # process event: openPanel
071                 # nothing to do.  We don't get notified of this event
072
073                 # process event: closePanel
074                 while not event.typeIs(OpenDoor, ClosePanel):
075                     event = yield()         # ignore irrelevant events
076                 if event.typeIs(OpenDoor): jsd.quit()      # the RESET event
077                 sendEndCommands()                          # to devices
078                 event = yield()                            # JSD getNextEvent
079
080             #-------------------------------------------------------------
081             # JSD "ADMIT" that we have an "interrupted access"
082             #-------------------------------------------------------------
083             except jsd.Quit:
084                 sendEndCommands()                          # to devices
085                 event = yield()                            # JSD getNextEvent
```

I have tried to comment the code so you can easily map chunks of code back to their corresponding nodes in the action structure diagram.

There are some pieces of code that don't correspond to anything on the STD or the action structure diagram. These are the bits of code that implement things that were left off of the diagrams.

- The transitions for the *openDoor* reset event.
- The circular transitions that don't change the controller's state.

Consider this code snippet:

```
043 # process event: closeDoor
044 while not event.typeIs(OpenDoor, CloseDoor):
045     event = yield()                         # ignore irrelevant events
046 if event.typeIs(OpenDoor): jsd.quit()      # the RESET event
```

Lines 44 and 45 implement a *while* loop that loops, ignoring all events that don't change the controller's state, until it finds some type of event that it wants to use.

Line 46 corresponds to the "reset events" on Fowler's STD. It is the code that triggers the JSD quit when a "reset" event (*openDoor*) is encountered.

```
if event.typeIs(OpenDoor):
       jsd.quit()
```

The summing up

I said in the introduction to this paper, that I would be presenting an argument-by-example for JSD event-oriented specifications. And I said that you could decide for yourself what you think of it.

My argument-by-example is now concluded. We have a JSD specification for Miss Grant's controller.

Now it is your turn.

- Compare the conceptual vocabulary of the state-oriented and the event-oriented approaches – states and state transitions vs. events and the order in which they may occur.

  Which do you prefer? Which do you find more intuitive?

- Compare the diagramming techniques – state transition diagrams vs. action structure diagrams.

  As you look at them, which do you think gives you a better overall understanding of the behavior of Miss Grant's controller?

- Compare the code.

  Look at the Python code for Miss Grant's controller, and then to compare it to the several different flavors of state-oriented specifications that Fowler provides on the Web page that I mentioned earlier.

  http://www.informit.com/articles/article.aspx?p=1592379&seqNum=3

What do you think of it?

The executable specifications executed

Wait! There's more! This is the fun stuff!

I claimed earlier that I had created an executable specification in Python. You've seen the code. Now it is time for me to prove that it really is executable.

To run the model, we need to create some test data – a stream of sensor events – that we can feed to the controller. So here is the code for the Python driver program. It creates a sequence of event objects and feeds them to the Python specification for Miss Grant's controller.

Note that we create the controller object by instantiating the MissGrantsController class. And then we set its *verbose* attribute to True. This tells the controller to log its activities as it runs.

```
001 from missGrantsController import *
002
003 #---------------------------------------
004 # Make the controller model process
005 #---------------------------------------
006 controller = MissGrantsController()
007
008 # tell the controller to log its activities
009 controller.verbose=True
010
011 #---------------------------------------
012 # Send events to the controller
013 #---------------------------------------
014 # normal access
015 controller.send( CloseDoor()     )
016 controller.send( TurnOnLight()   )
017 controller.send( OpenDrawer()    )
018 controller.send( ClosePanel()    )
019
020 # interrupted access
021 controller.send( CloseDoor()     )
022 controller.send( TurnOnLight(1)  )
023 controller.send( TurnOnLight(2)  )
024 controller.send( TurnOnLight(3)  )
025
026 controller.send( ClosePanel("contextError"))
027 controller.send( ClosePanel("contextError"))
028
029 controller.send( OpenDrawer()    )
030 controller.send( OpenDoor ("forces QUIT")  )
031 controller.send( ClosePanel("contextError"))
```

Finally, here is the output produced by a test run.

Even in *verbose* mode, the original version of Miss Grant's controller would produce only the lines beginning with EVENT and the lines reporting the *lock* and *unlock* commands. I wanted to be able to show you more details of what the controller was doing while it executed, so this is the output of a version of the controller that I enhanced with a few logging commands.

```
C:/Python30/python.exe C:/pydev/pyJSD/missGrantsController_run.py
EVENT (   1) MissGrantsController() got event: CloseDoor()
   Yes! processing event: CloseDoor
EVENT (   2) MissGrantsController() got event: TurnOnLight()
   Yes! processing event: TurnOnLight
EVENT (   3) MissGrantsController() got event: OpenDrawer()
   Yes! processing event: OpenDrawer
   sendStartCommands
   door  locked
   panel unlocked
EVENT (   4) MissGrantsController() got event: ClosePanel()
   Yes! processing event: ClosePanel
   sendEndCommands
   panel locked
   door  unlocked
EVENT (   5) MissGrantsController() got event: CloseDoor()
   Yes! processing event: CloseDoor
EVENT (   6) MissGrantsController() got event: TurnOnLight(1)
   Yes! processing event: TurnOnLight
EVENT (   7) MissGrantsController() got event: TurnOnLight(2)
   Yes! processing event: TurnOnLight
EVENT (   8) MissGrantsController() got event: TurnOnLight(3)
   Yes! processing event: TurnOnLight
EVENT (   9) MissGrantsController() got event: ClosePanel(contextError)
   **** ignoring   event: ClosePanel
EVENT (  10) MissGrantsController() got event: ClosePanel(contextError)
   **** ignoring   event: ClosePanel
EVENT (  11) MissGrantsController() got event: OpenDrawer()
   Yes! processing event: OpenDrawer
   sendStartCommands
   door  locked
   panel unlocked
EVENT (  12) MissGrantsController() got event: OpenDoor(forces QUIT)
   Yes! processing event: OpenDoor
   sendEndCommands
   panel locked
   door  unlocked
EVENT (  13) MissGrantsController() got event: ClosePanel(contextError)
   **** ignoring   event: ClosePanel
```

The primary purpose of creating executable specifications is so that we can actually compile, run, test, and debug our specifications.

I can testify that it works. I didn't get MissGrantsController.py right the first time. There were indeed bugs. But I executed the specifications, I found the bugs, and I fixed them.

That's what it's all about.

Prior art: Python Coroutines

Python Enhancement Proposal (PEP) 342: Coroutines via Enhanced Generators (10-May-2005)

- http://www.python.org/dev/peps/pep-0342/
- http://docs.python.org/whatsnew/2.5.html#pep-342-new-generator-features

David Beazley

- talks and presentations at www.dabeaz.com/talks.html, especially...
- *A Curious Course on Coroutines and Concurrency* at www.dabeaz.com/coroutines

  Video of the presentation, in 3 parts:
  http://python.mirocommunity.org/video/996/pycon-2009-a-curious-course-on
  http://python.mirocommunity.org/video/994/pycon-2009-a-curious-course-on
  http://python.mirocommunity.org/video/995/pycon-2009-a-curious-course-on

Short papers by David Mertz at IBM DeveloperWorks

- Generator-based state machines
  http://www.ibm.com/developerworks/library/l-pygen.html
- Implementing "weightless threads" with Python generators
  http://www.ibm.com/developerworks/library/l-pythrd.html

SimPy (= ***Sim****ulation in* ***Py****thon*) is an object-oriented, process-based discrete-event simulation language based on standard Python. Basically, SimPy wraps coroutines in syntactic sugar in order to make it easier to use them to do discrete event simulation.

- http://simpy.sourceforge.net/
- http://www.ibm.com/developerworks/linux/library/l-simpy.html
- http://heather.cs.ucdavis.edu/~matloff/simpy.html
- http://heather.cs.ucdavis.edu/~matloff/156/PLN/DESimIntro.pdf

Stackless Python is a Python implementation that supports a lot of neat stuff that CPython does not, including very nice support for coroutines.

- http://www.stackless.com/
- http://en.wikipedia.org/wiki/Stackless_Python
- http://www.stackless.com/spcpaper.pdf