Chapter 3

Developing a SimPy simulation program

In this chapter, the process and tools for developing a simulation program in SimPy will be discussed. In order not to distract from the approach, the model/scenario used is very simple, namely the bank scenario from Chapter 1.

Basic structure of a SimPy program

To implement a simulation model in SimPy, the user has to program the following steps in Python plus the SimPy framework facilities:

1. import the SimPy simulation library,
2. define at least one class of active simulation components (processes),
3. formulate a model which:
4. initializes the SimPy runtime machinery (effectively, the event handler),
5. generates at least one or more instances of his classes of active components,
6. activates those instances,
7. sets up data collection,
8. starts the simulation's execution,
9. set experiment data values for a simulation experiment
10. run the experiment
11. analyze the collected data,
12. output the results

This can be mapped into a SimPy program structure like this:

```
import SimPy.Simulation as Sim            (1)
## Model components
class MyProcess(Sim.Process):             (2)
    def myPEM(self):
       yield hold,self,myDuration
## Model
def model():                              (3)
   Sim.initialize()                       (4)
   p=MyProcess()                          (5)
   Sim.activate(p,p.myPEM(),at=tFirst)    (6)
   results=Sim.Monitor()                  (7)
   Sim.simulate(until=endtime)            (8)
   return results
## Experiment data                        (9)
tFirst=100
myDuration=10
endtime=1234
# Experiment
outcome=model()                           (10)
# Analysis
answer=outcome.mean()                     (11)
# Output
print “Result is: %s”%answer              (12)
```

The author has found it very useful to have the same structure with six sections for all SimPy simulation programs:

```
## Model components
## Model
## Experiment data
## Experiment
## Analysis
## Output
```

If a simulation program has model runs (experiments) using several data sets, the last four sections should be repeated as many times as there are data sets.

Such canonical structure makes it easy for the program developer to find his way in his own programs or those of others. This structure will be used in all programs presented in this book.

It is highly advisable to keep function `model` free of literal data and just use variables which are assigned their values in a set of experiment data. This keeps the model general and allows it to be used for any number of experiments (simulation runs), based on different experiment data sets.

Similarly, all model components should be kept free of data by using variables which get their values from the experiment data, either directly or by parameter transmission from function `model`.

The objects qualified by `Sim` are provided by the `SimPy.Simulation` module. They give the user program these capabilities:

- By inheriting from `Sim.Process`, instances of class `MyProcess` can become active components for which events can be scheduled. The PEM `myPEM` defines their lifecycle.
- `Sim.initialize` sets up the simulation runtime machinery. It sets up an event list and sets the initial simulation time to 0.
- The `Sim.activate` call schedules an initial event for the `MyProcess` entity `p` at time `tFirst`.
- The `Sim.simulate` call executes all events scheduled on the event list in time sequence until either no more events are scheduled or the simulation clock has reached time `endtime`.

There are many more components provided by `SimPy.Simulation`. They will be presented later in this book.

`SimPy.Simulation` is just one of the four available simulation library modules. It is used in most cases. Other, alternative simulation library modules available are:

`SimPy.SimulationTrace`:

Provides all of SimPy.Simulation's capabilities, plus extensive execution tracing features. Ideal for model introspection for debugging, teaching or documentation purposes.

`SimPy.SimulationStep`:

All of SimPy.Simulation's capabilities, plus the capability to execute a model event by event. This is useful for e.g. interactive simulations with a man in the loop, interacting with the model.

`SimPy.SimulationRT`:

All of SimPy.Simulation's capabilities, plus the capability to synchronize simulation time with wall clock time. This is useful for e.g. implementing models of physical systems where the simulation user is to get insight into the timing of events, the dynamic behaviour of the system modeled.

Only one of the four libraries may be invoked at any one time in a SimPy program.

The user new to SimPy should always import the SimPy facilities by `import SimPy.Simulation as <name>`, e.g. `import SimPy.Simulation as Sim`. In the model, she then has to fully qualify all imported objects, e.g. `Sim.Process`, **or** `Sim.hold`. This quickly teaches her which constructs come from

the SimPy library and also protects her from accidentally overloading them with her own objects. Later, once she is familiar with SimPy, the amount of writing may be reduced by using `from SimPy.Simulation import *` and writing `Process` or `hold`.

Tools to use

Good tools help a worker in any field with becoming more productive. If you are a beginning programming SimPy user, your minimum set of tool capabilities available should be:

- An editor with Python syntax highlighting, preferably one which supports program syntax checking and execution. It should also have code folding so that you can collapse lower level blocks. Examples of free tools are SciTE (multi-platform), IDLE (multi-platform), Stani’s Python Editor (MS Windows, Linux, Mac), PythonWin IDE (MS Windows).
  See http://wiki.python.org/moin/PythonEditors for a list.
- A web browser with a bookmark set to the SimPy documentation (on the local computer or at http://simpy.sourceforge.net/discuss.htm.) As you work on a SimPy program, you will have easy access to reference documents, manuals, tutorials, sample models and source code documentation.

When larger, complex SimPy programs are to be developed or studied, there is one advanced tool which can help a lot:

- An integrated development environment (IDE) for Python with a built-in debugging capability for stepping through a SimPy script, setting breakpoints, tracking variable values. It should also have an object browser which allows browsing the object hierarchy (classes, functions). Examples are IDLE, PythonWin, SPE.
  See http://wiki.python.org/moin/IntegratedDevelopmentEnvironments for a list.

Development approach

I recommend an incremental, iterative design—code—run approach (also called “build a little, run a little, learn a lot”).

Your design should start with a written scenario of the situation to be simulated. In that scenario, you should identify actors, their actions and the resources they need. In your design, write down the lifecycles of your actors. My preference is to write the lifecycle as Python-like pseudo-code.

Most importantly, you should define what results you want to get out of your simulation.

As soon as you have this first level of understanding, you should start to code the top level of your program, using comments saying what code is to do and stubs standing in for code yet to be written. You will be surprised by how much this will force you to re-think and elaborate your model. Use the

```
# Model components
# Model
# Experiment data
# Experiment
# Analysis
# Output
```

structure to lay out your program.

In the early phases of your program evolution, use

```
import SimPy.SimulationTrace as Sim
```

or

```
from SimPy.SimulationTrace import *
```

to import the SimPy library, rather than `SimPy.Simulation`. This will give you insight into processes, their events and interactions, resources, etc. by providing a trace of all scheduling commands. Later, when

you are confident that your program structure is correct, you can switch off the tracing by replacing `SimPy.SimulationTrace` by `SimPy.Simulation`.

Familiarize yourself with your debugger in your IDE. It will become your best friend! Professor Norm Matloff’s great SimPy tutorial

http://heather.cs.ucdavis.edu/~matloff/simpy.html

has a very helpful discussion on how to use a debugger for debugging SimPy programs.

If there are conditions which must be true at a certain point in your program, put them into your code using Python’s `assert` statement. Use them liberally, as they make your design intentions operational in your program. They are great for ensuring correct synchronization between multiple parallel processes. For example, if the number of widgets (say, `nWidgets`) in existence should never exceed 3 at a certain point in the program, write there

```
assert nWidgets<=3, “Too many widgets. Nr of widgets is %s”%nWidgets
```

Should this design condition of yours get violated, you will be told by an assertion error message. Leave your `assert` statements in your final program version, too! They are ignored if you don’t run Python in debug mode, so they do not cause any runtime overhead.

Let’s build our first SimPy program

The scenario we are going to use is very simple, you know it already: the bank visit example from the Introduction chapter. Here is the scenario description again:

A bank opens its beautiful new branch office and invites the public to an Open Day. The visitors arrive, are greeted by the branch manager one by one and can then look around in the new office. They then leave.

To get some insight into this situation, let us follow the events that happen to three visitors (Tony, Klaus and Simon) who arrive at 0, 0.5 and 1 minutes after opening, respectively. The branch manager spends exactly 1 minute with each visitor. A visitor looks around for 10 minutes.

Let’s add a result we are looking for:

At what time have all three visitors left the bank?

Now let’s identify the actors:

- Visitor Tony
- Visitor Klaus
- Visitor Simon
- Bank manager

The visitors all have the same lifecycle:

- Arrive
- Try to get free manager; wait if none available
- Engage the manager (make him busy)
- Talk to the manager for 1 minute
- Free the manager (make him not busy)
- Look around for 10 minutes
- Leave

There is still another component to consider (implicit in the scenario):

- The bank which the visitors visit and to whom the manger belongs

And the scenario data is:

- Arrival times: Tony/time 0, Klaus/time 0.5, Simon/time 1.0 (all in minutes)
- Talk time: 1 minute
- Look time: 10 minutes

Now it comes to the program design decisions:

- As the visitors are active components, they are modeled as *entities*.
- As these three visitor entities have the same life cycle and just differ in their data (name, arrival time), they are modeled as instances of a class `Visitor`, with a Process Execution Method `visit`.
- The bank manager does not really have a lifecycle. The visitors just make him busy and then not busy again. He could be modeled as an entity, but it is easier and sufficient to just model him by a data structure, as an instance of a class `Manager`.

Let’s fire up our editor and type in a first outline program:

```
1   """bank-v0.py"""
2   import SimPy.SimulationTrace as Sim
3   ## Model components
4   class Visitor(Sim.Process):
5       def visit(self,bank):
6           pass
7           # try to get free manager; wait if none available
8           # make manager busy
9           # talk to manager
10          # free manager for other visitors
11          # look around
12          # leave
13  class Manager: pass
14  class Bank: pass
15  ## Model
16  def model():
17      Sim.initialize()
18      myBank=Bank()
19      myBank.manager=Manager()
20      for visname,arrivaltime in visitorlist:
21          vis=Visitor(name=visname)
22          Sim.activate(vis,vis.visit(bank=myBank),at=arrivaltime)
23      Sim.simulate(until=simtime)
24  ## Experiment data
25  visitorlist=[("Tony",0),("Klaus",0.5),("Simon",1.0)]
26  tChat=1; tLook=10
27  simtime=100
28  ## Experiment
29  model()
30  ## Analysis/output
31  print "All visitors gone at time %s"%Sim.now()
```

You don’t type in the line numbers. They are just given for reference purposes. Many editors can actually generate line number automatically. They are useful for identifying code lines where errors have occurred—error messages provide both context and line numbers of error locations.

What do the various lines of code do?

Line	Description
1	I always include the file name. The first string in a Python program is the program’s doc string which can be accessed under the name `__doc__`. This can be useful e.g. for inclusion in the program’s output, to identify where it came from.
2	Makes the SimPy constructs available to the program. When you start development of a SimPy program, import from `SimulationTrace`

Line	Description
	(instead from `Simulation`) to get tracing of events. It will help you!
4	The `Visitor` class, derived from SimPy class `Process`. This allows Visitor instances to be entities, active components.
5..12	Class `Visitor`’s `visit` PEM, prescribing the lifecycle of `Visitor` entities. It has the parameter `bank` to identify which bank is actually visited.
6	Function `visit` is just still a stub, so `pass` is the only statement in the PEM
7..12	The pseudo-code, yet to be turned into SimPy statements
13	The class for the Manager; just a stub, so `pass`
14	The class for the bank; just a stub, so `pass`
16..22	The model called `model`, based on the model components. You could call this function anything, the name does not matter.
17	Initialize the SimPy simulation machinery
18	Generate one `Bank` instance
19	Give the `Bank` instance one `Manager` instance
20	Step through the `visitorlist` model data list which contains the tuples (name, arrival time)
21	For each tuple, create a `Visitor` object and give it the name from the data tuple. If you wonder why there is no declaration of a `name` attribute in class `Visitor`: this attribute is inherited from class `Process`—all `Process` objects have a `name` field.
22	Activate the `Visitor` entity just created at the arrival time found in the data tuple. This puts a first (kick-off) event for this entity on the event list. The entity is passive until that time.
23	Start the simulation. The `until` parameter sets the maximum simulated time. At that time, the model execution ends, even if processes are still active. The simulation can actually end at an earlier time if all processes are terminated, a call to `stopSimulation` is issued, or a SimPy error situation occurs.
25..27	The data set for this specific simulation experiment. It provides the data for the model components and the model. Change any of this, and you have a different experiment.
29	The call to start the model execution.
31	There is no separate results analysis; the result is just the simulation time at the end of the simulation experiment. That is what output is. This is clearly a weak design—what if the simulation ends before all visitors have completed their lifecycle, because of an error?

At all times during its development, such a program should be runable (even if its results may still be meaningless). So, let’s try it:

```
>>> python bank_v1.py
Traceback (most recent call last):
  File "bank_v0.py", line 29, in ?
    model()
```

```
  File "bank_v0.py", line 22, in model
    Sim.activate(vis,vis.visit(bank=myBank),at=arrivaltime)
  File "C:\Python23\lib\site-packages\SimPy\SimulationTrace.py", line
359, in activate
    raise Simerror("Fatal SimPy error: activating function which"+
SimPy.SimulationTrace.Simerror: "Fatal SimPy error: activating function
which is not a generator (contains no 'yield')"
```

Oops! This message tells us that a PEM (which is a Python `generator`), in this case the PEM `visit`, must have at least one `yield` scheduling command. Without it, Python just treats `visit` as a normal function, and `SimPy.Simulation.Trace` reports the error that you can’t activate a function. So, even a stub of a PEM must always have one or more `yield` statements.

To make the program run without problems, we can just insert the activation statements for the “talk to manager” and “looking around” activities of `Visitor` entities behind the appropriate pseudo-code comments (lines 9 and 11) in `visit`. While we are at it, we realize that we have a condition which must be true at the end of the simulation: the number of visitors which have completed their lifecycle must be equal to the number which were generated and activated. We therefore add a class variable `nrDone` to `Visitor`, increment it at the end of the PEM, and assert the condition after the simulate statement. We now have:

```
32  """bank_v1.py"""
33  import SimPy.SimulationTrace as Sim
34  ## Model components
35  class Visitor(Sim.Process):
36      nrDone=0
37      def visit(self,bank):
38          # try to get free manager; wait if none available
39          # make manager busy
40          # talk to manager
41          yield Sim.hold,self,tChat
42          # free manager for other visitors
43          # look around
44          yield Sim.hold,self,tLook
45          # leave
46          Visitor.nrDone+=1
47  class Manager: pass
48  class Bank: pass
49  ## Model
50  def model():
51      Sim.initialize()
52      myBank=Bank()
53      for visname,arrivaltime in visitorlist:
54          vis=Visitor(name=visname)
55          Sim.activate(vis,vis.visit(bank=myBank),at=arrivaltime)
56      Sim.simulate(until=simtime)
57      assert Visitor.nrDone==len(visitorlist),\
58             "wrong nr of visitors are done"
59  ## Experiment data
60  visitorlist=[("Tony",0),("Klaus",0.5),("Simon",1.0)]
61  tChat=1; tLook=10
62  simtime=100
63  ## Experiment
64  model()
65  ## Analysis/output
66  print "All visitors gone at time %s"%Sim.now()
```

Run this, and you get:

```
>>> python bank_v1.py
```

```
0 hold <Tony> delay: 1
0.5 hold <Klaus> delay: 1
1.0 hold <Simon> delay: 1
1.0 hold <Tony> delay: 10
1.5 hold <Klaus> delay: 10
2.0 hold <Simon> delay: 10
11.0 <Tony> terminated
11.5 <Klaus> terminated
12.0 <Simon> terminated
All visitors gone at time 12.0
```

It works (but gives the wrong result, of course)! For each of the three entities, `SimulationTrace` traces the activities in their (intertwined) lifecycles. The first activity is the talking to the manager and the second the looking around. The third entry for each entity is the termination event, i.e., when they exhaust their lifecycle.

The error in this still incomplete program is that the talk phases of Klaus and Tony and of Simon and Klaus overlap. This means that the manager is talking to more than one visitor at a time, a violation of a key design specification. We should have put this into the program as an `assert` statement right from the beginning.

Let’s repair this omission now. We give the manager an attribute `nrTalking`, the number of visitors he is talking to, and initialize it to zero. We increment it at the beginning of the talk phase and decrement it at the end of the talk phase. The `assert` statement is put right behind the increment of `nrTalking`.

The new version of our program is:

```
"""bank_v3.py"""
import SimPy.SimulationTrace as Sim
## Model components
class Visitor(Sim.Process):
    nrDone=0
    def visit(self,bank):
        # try to get free manager; wait if none available
        # make manager busy
        bank.manager.nrTalking+=1
        assert bank.manager.nrTalking==1,\
        "At %s: Manager is talking to more than one visitor"%Sim.now()
        # talk to manager
        yield Sim.hold,self,tChat
        # free manager for other visitors
        bank.manager.nrTalking-=1
        # look around
        yield Sim.hold,self,tLook
        # leave
        Visitor.nrDone+=1
class Manager:
    def __init__(self):
        self.nrTalking=0
class Bank: pass
## Model
def model():
    Sim.initialize()
    myBank=Bank()
    myBank.manager=Manager()
    for visname,arrivaltime in visitorlist:
        vis=Visitor(name=visname)
```

```
            Sim.activate(vis,vis.visit(bank=myBank),at=arrivaltime)
        Sim.simulate(until=simtime)
        assert Visitor.nrDone==len(visitorlist),"wrong nr of visitors are done %s"%Visitor.nrDone
    ## Experiment data
    visitorlist=[("Tony",0),("Klaus",0.5),("Simon",1.0)]
    tChat=1; tLook=10
    simtime=100
    ## Experiment
    model()
    ## Analysis/output
    print "All visitors gone at time %s"%Sim.now()
```

When we run this version, we get this output:

```
>>> python bank_v3.py
0 hold <Tony> delay: 1
Traceback (most recent call last):
  File "bank_v3.py", line 39, in ?
    model()
  File "bank_v3.py", line 32, in model
    Sim.simulate(until=simtime)
  File "C:\Python23\lib\site-packages\SimPy\SimulationTrace.py", line
1484, in simulate
    a=_e._nextev()
  File "C:\Python23\lib\site-packages\SimPy\SimulationTrace.py", line
336, in _nextev
    tt=tempev.who._nextpoint.next()
  File "bank_v3.py", line 10, in visit
    assert bank.manager.nrTalking==1,\
AssertionError: At 0.5: Manager is talking to more than one visitor
```

As we suspected, the program does not meet out design specifications yet. The `AssertionError` reports that at time 0.5 (when visitor Klaus starts talking to the Manager), more than one visitors are talking to him. Of course, Tony is still talking to him. This is a *synchronization issue* which we must tackle now. Finally, we get into inter-process synchronization.

The part of the visitor lifecycle from “make manager busy” and “free manger for other visitors” is a critical region in which at any one time, there must be at most one visitor entity. We must therefore guard this region so that no visitor entity can enter that region if another entity is already in it.

The pseudo-code “try to get free manager” (line 7) can be implemented by testing for the number of visitors the manager is talking to. The “wait if none available” (l. 7) requires a place where a visitor can wait. We make that a queue for the manager which we implement as a list attribute `visitorQ` for the manager entity. The waiting is then implemented as joining the queue (`append`) and making the visitor entity passive by a `yield Sim.passivate` command.

The “free manager for other visitors” (l. 14) must also take care of reactivating the first visitor in the `visitorQ`. This is done by checking for there being someone waiting. If the there is, the first visitor entity is taken out of the `visitorQ` and reactivated.

These evolutionary additions give us the following program:

```
    """bank_v5.py"""
    import SimPy.SimulationTrace as Sim
    ## Model components
```

```
    class Visitor(Sim.Process):
        nrDone=0
        def visit(self,bank):
            # try to get free manager; wait if none available
            if bank.manager.nrTalking>0:
                bank.manager.visitorQ.append(self)
                yield Sim.passivate,self
            # make manager busy
            bank.manager.nrTalking+=1
            assert bank.manager.nrTalking==1,\
            "At %s: Manager is talking to more than one visitor"%Sim.now()
            # talk to manager
            yield Sim.hold,self,tChat
            # free manager for other visitors
            bank.manager.nrTalking-=1
            # continue first waiting visitor
            if len(bank.manager.visitorQ)>0:
                nextVisitor=bank.manager.visitorQ.pop(0)
                Sim.reactivate(nextVisitor)
            # look around
            yield Sim.hold,self,tLook
            # leave
            Visitor.nrDone+=1
    class Manager:
        def __init__(self):
            self.nrTalking=0
            self.visitorQ=[]
    class Bank: pass
    ## Model
    def model():
        Sim.initialize()
        myBank=Bank()
        myBank.manager=Manager()
        for visname,arrivaltime in visitorlist:
            vis=Visitor(name=visname)
            Sim.activate(vis,vis.visit(bank=myBank),at=arrivaltime)
        Sim.simulate(until=simtime)
        assert Visitor.nrDone==len(visitorlist),"wrong nr of visitors are done"
    ## Experiment data
    visitorlist=[("Tony",0),("Klaus",0.5),("Simon",1.0)]
    tChat=1; tLook=10
    simtime=100
    ## Experiment
    model()
    ## Analysis/output
    print "All visitors gone at time %s"%Sim.now()
```

Let’s hold our breath and run this:

```
>>> python bank_v5.py
0 hold <Tony> delay: 1
0.5 passivate <Klaus>
1.0 passivate <Simon>
1.0 hold <Tony> delay: 10
1.0 hold <Klaus> delay: 1
2.0 hold <Klaus> delay: 10
2.0 hold <Simon> delay: 1
3.0 hold <Simon> delay: 10
```

```
11.0 <Tony> terminated
12.0 <Klaus> terminated
13.0 <Simon> terminated
All visitors gone at time 13.0
```

Eureka, this works! The last line is our simulation experiment’s resutlt, the last of the three visitors leaves the bank 13 minutes after the start.

Clearly, the most complex and error-prone part of this (or any other) simulation program is that dealing with the inter-process synchronization. SimPy makes the implementation of correct simulation programs much easier by providing higher-level constructs for typical inter-process synchronizations.

Just to give you a preview (much more to come in the remainder of the book): In our little bank example, the manager is a *resource*, a choke point, where visitor entities may have to queue. The lifecycle part where an entity uses it is a critical region (only one allowed inside). These constructs of resources and critical regions are so frequently used in simulation models that SimPy introduced the high level constructs class `Resource` and two synchronization commands to request (`yield request`) and to release (`yield release`) a resource. Using them, you get a much shorter program with correct synchronization “out of the box”:

```
"""bank_highlevel.py"""
import SimPy.SimulationTrace as Sim
## Model components
class Visitor(Sim.Process):
    nrDone=0
    def visit(self,bank):
        # try to get free manager; wait if none available
        # make manager busy
        yield Sim.request,self,bank.manager
        assert len(bank.manager.activeQ)==1,\
        "At %s: Manager is talking to more than one visitor"%Sim.now()
        # talk to manager
        yield Sim.hold,self,tChat
        # free manager for other visitors
        # continue first waiting visitor
        yield Sim.release,self,bank.manager
        # look around
        yield Sim.hold,self,tLook
        # leave
        Visitor.nrDone+=1
class Bank: pass
## Model
def model():
    Sim.initialize()
    myBank=Bank()
    myBank.manager=Sim.Resource(name="Managers",capacity=1)
    for visname,arrivaltime in visitorlist:
        vis=Visitor(name=visname)
        Sim.activate(vis,vis.visit(bank=myBank),at=arrivaltime)
    Sim.simulate(until=simtime)
    assert Visitor.nrDone==len(visitorlist),"wrong nr of visitors are done"
## Experiment data
visitorlist=[("Tony",0),("Klaus",0.5),("Simon",1.0)]
tChat=1; tLook=10
simtime=100
## Experiment
model()
## Analysis/output
print "All visitors gone at time %s"%Sim.now()
```