17 More Testing: Non-terminating Process Networks

Chapter 6 showed it is possible to use the `GroovyTestCase` capability to test networks of processes, provided each of the processes in the network terminates. Most of the processes used in this book do not terminate and so a means of testing such non-terminating process networks has to be developed.

First, however, we need to reflect on the operation of `PAR`. A `PAR` only terminates when all the process in the list of processes passed to it terminate. Thus, if only one of the processes does not terminate then the PAR will never terminate. However, if the assertion testing commonly used in `JUnit` and `GroovyTestCase` is to be undertaken then at least some of the test environment has to terminate. Figure 10-1 shows a generic architecture that allows a process network under test (PNUT) to run without terminating, while the Test-Network does terminate, which then allows the assertion testing to take place in the normal manner.

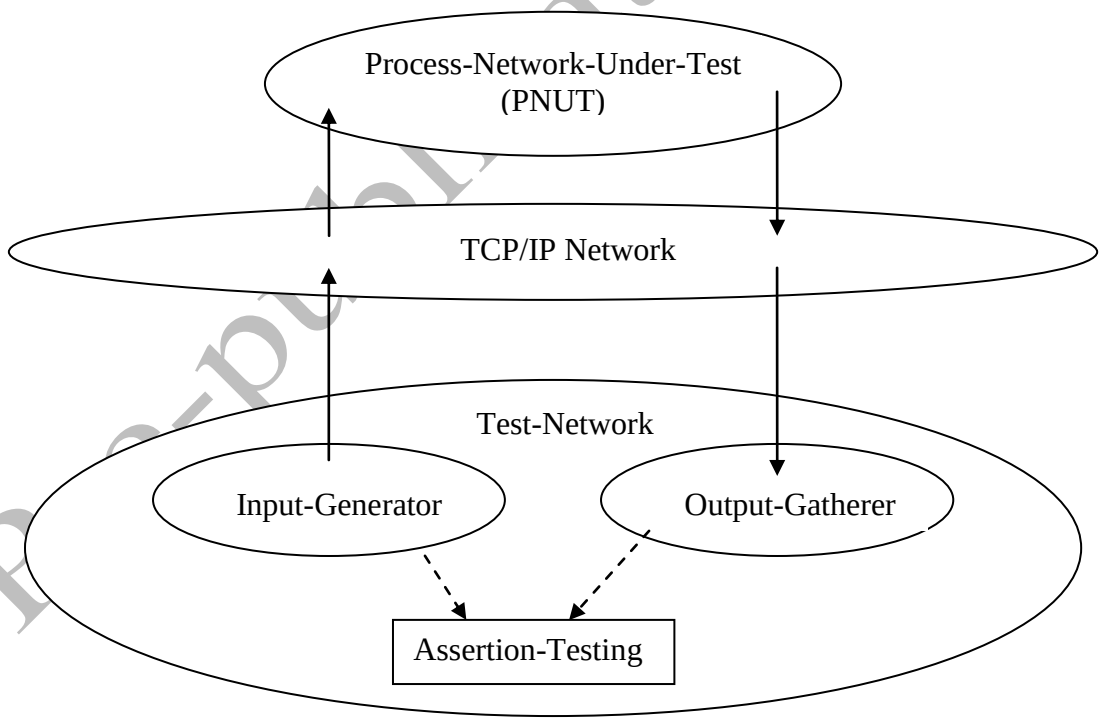

Figure 17-1 Generic Testing Architecture

The separation of the PNUT from the Test-Network by means of a TCP/IP communications network means that the two process networks run independently of each other and it does not matter if the PNUT

does not terminate, provided the Test-Network does. We can assume that the PNUT requires input and also that it outputs results in some format. This data is communicated by means of the network channels shown. Both the Input-Generator and Output-Gatherer processes must run as a PAR within the process Test-Network, then terminate; after which their internal data structures can be tested within Assertion-Testing. This demonstrates the generic nature of the architecture in that the only part that has to be specifically written is the processes that implement the Input-Generator and Output-Gatherer respectively. The architecture will now be demonstrated using the Scaling Device example described previously in Chapter 6. The Scaling Device takes a stream of input numbers and outputs an equivalent stream of scaled numbers, while monitoring the operation of a Scale process by modifying the applied scaling factor.

17.1 The Test-Network

The class RunTestPart, shown in Listing 17-1, implements the Test-Network {1} and simply extends the class GroovyTestCase. The method testSomething {2} creates the Test-Network as a process running in a node on a TCP/IP network. The node is initialized in the normal manner within the JCSP framework {3}. Two NetChannels, ordinaryInput {4}and scaledOutput {5} are defined and recorded within an instance of TCPIPCNSServer that is presumed to be running on the network prior to the invocation of both the PNUT and Test-Network. The processes are created {6, 7} and then invoked {8, 9}. Once the PAR has terminated, the properties generatedList, collectedList and scaledList can be obtained from the processes {10-12} using the Groovy dot notation for accessing class properties. In this case we know that the original generated set of values should equal the unscaled output from the collector and this is tested in an assertion {13}. In this case we also know that each modified output from the PNUT should be greater than or equal to the corresponding input value. This is implemented by a method contained in a package TestUtilities called list1GEList2, which is used in a second assertion {14}.

```
01    class RunTestPart extends GroovyTestCase {

02       void testSomething() {

03          Node.getInstance().init(new TCPIPNodeFactory ())

04          NetChannelOutput ordinaryInput = CNS.createOne2Net("ordinaryInput")
05          NetChannelInput scaledOutput = CNS.createNet2One("scaledOutput")

06          def collector = new CollectNumbers ( inChannel: scaledOutput)
07          def generator = new GenerateNumbers (outChannel: ordinaryInput)

08          def testList = [ collector, generator]

09          new PAR(testList).run()

10          def original = generator.generatedList
11          def unscaled = collector.collectedList
12          def scaled = collector.scaledList
13          assertTrue (original == unscaled)
14          assertTrue (TestUtilities.list1GEList2(scaled, original))
15       }
16    }
```

Listing 17-1 The Extended GroovyTestCase Class to Run The Test Network

The benefit of this approach is that we are guaranteed that the Test-Network will terminate, provided the CollectNumbers and GenerateNumbers processes terminate and thus values derived from these processes can be tested in assertions. The fact that the PNUT continues running is made disjoint by the use of the TCP/IP network. This could not be achieved if all the processes were run in a single JVM as the assertions could not be tested because the PAR would never terminate. The process network comprising the PNUT and the Test-Network can be run on a single processor with each running in a separate JVM, as is the TCPIPCNSServer. RunTestPart will write its output to a console window indicating whether or not the test has passed. The console window associated with PNUT will continue to produce any outputs associated with the network being tested.

17.1.1 The Generate Numbers Process

Listing 17-2 shows the coding of the `GenerateNumbers` process. Recall from Chapter 6 that the Scaling Device expects to receive numbers at regular intervals, which it then processes. The property `delay` {18} is used to specify the time between the generation of an output of numbers to the PNUT. The length of the generated sequence is specified in `iterations` {19}. The channel `outChannel` {20} is used to communicate the generated numbers to the PNUT. The list `generatedList` {21} is used to hold the sequence of generated numbers for subsequent testing in an assertion {10}.

```
17    class GenerateNumbers implements CSProcess{

18      def delay = 1000
19      def iterations = 20
20      def ChannelOutput outChannel

21      def generatedList = []

22      void run() {
23        println "Numbers started"
24        def timer = new CSTimer()
25        for (i in 1 .. iterations) {
26          outChannel.write(i)
27          generatedList << i
28          timer.sleep(delay)
29        }
30        println "Numbers finished"
31      }
32    }
```

Listing 17-2 The GenerateNumbers Process

The `run` method {22} outputs a start message {23} and then defines a `timer` {24}. Each number is then generated using a `for` loop {25}, limited by the value of `iterations`. The next number in sequence is output {26} and then appended (<<) to `generatedList` {27}. The process then `sleeps` for the defined `delay` period {28}. Finally, a finished message is output {30}.

17.1.2 The Collect Numbers Process

Listing 17-3 shows the `CollectNumbers` process. The `inChannel` {34} is used to input data from the PNUT. The output from the Scaling Device is in the form of objects comprising two properties; the `original` value and the `scaled` value (See 6.1.3). The `original`, unmodified values are appended {42} to the property `collectedList` {35} and the `scaled` values are appended {43} to the property `scaledList` {36}. The number of `iterations` {37} is required to ensure that the process terminates after it has read the expected number of outputs from the PNUT. The `run` method simply iterates over the expected outputs, inputting `ScaledData` from the PNUT {41} and placing the data into the respective lists {42, 43}. The method also indicates, by console messages, when the process starts {40} and finishes {45}.

```
33    class CollectNumbers implements CSProcess {

34      def ChannelInput inChannel
35      def collectedList = []
36      def scaledList = []
37      def iterations = 20

38      void run() {
39        println "Collector Started"
40        for ( i in 1 .. iterations) {
41          def result = (ScaledData) inChannel.read()
42          collectedList << result.original
43          scaledList << result.scaled
44        }
45        println "Collector Finished"
46      }
47    }
```

Listing 17-3 The CollectNumbers Process

17.2 The Process Network Under Test

The Process Network Under Test (PNUT) is the `ScalingDevice` described in Chapter 6 and can be represented by the `CSProcess` shown in Listing 17-4.

```
48    class ScalingDevice implements CSProcess {

49      def ChannelInput inChannel
50      def ChannelOutput outChannel

51      void run() {
52        def oldScale = Channel.createOne2One()
53        def newScale = Channel.createOne2One()
54        def pause = Channel.createOne2One()

55        def scaler = new Scale ( inChannel: inChannel,
56                                 outChannel: outChannel,
57                                 factor: oldScale.out(),
58                                 suspend: pause.in(),
59                                 injector: newScale.in(),
60                                 multiplier: 2,
61                                 scaling: 2 )

62        def control =  new Controller ( testInterval: 7000,
63                                        computeInterval: 700,
64                                        addition: 1,
65                                        factor: oldScale.in(),
66                                        suspend: pause.out(),
67                                        injector: newScale.out() )

68        def testList = [ scaler, control]
69        new PAR(testList).run()
70      }
71    }
```

Listing 17-4 The Scaling Device Process Definition

The `ScalingDevice` has an `inChannel` property {49} from which input numbers are read and an `outChannel` property {50} to which objects of `ScaledData` are written. The `run` method is simply the parallel instantiation {68, 69} of a `Scale` {55} and a `Controller` {62} process. These processes are connected by means of the channels `oldScale` {52}, `newScale` {53} and `pause` {54} as described in 6.1 and 6.1.4.

17.3 Running The Test

The test requires that the two parts, PNUT and TestNetwork execute as nodes of a network and therefore the processes have to be executed in conjunction with the `CNSServer`. The script to run the `ScalingDevice` node is shown in Listing 17-5. Prior to creating the `Node` {73} a call {72} is made to a static method of `Node`, `setDevice()` on a property `info`, which has the effect of eliminating the `CNSServer` generated output concerning the creating of network channels. This makes it easier to read the console output. Two network channels are created, `ordinaryInput` {74} and `scaledOutput` {75} that connect the `ScalingDevice` to the Test Network. The `ScalingDevice` process is then invoked within a PAR {76, 77}.

```
72    Node.info.setDevice(null)

73    Node.getInstance().init(new TCPIPNodeFactory ())

74    NetChannelInput ordinaryInput = CNS.createNet2One("ordinaryInput")
75    NetChannelOutput scaledOutput = CNS.createOne2Net("scaledOutput")

76    new PAR( new ScalingDevice ( inChannel: ordinaryInput,
77                                 outChannel: scaledOutput) ).run()
```

Listing 17-5 The ScalingDevice Node Script

The node that runs the TestNetwork is created as part of the class `RunTestPart`, Listing 17-1, where it can be seen that the corresponding ends of the channels, `ordinaryInput` {4} and `scaledOutput` {5} are created.

Typical output that appears on the console window associated with the RunTestPart node is shown in Output 17-1. The initial dot indicates that a test has been run and is generated automatically by the `GroovyTestCase`. The starting messages from both the `collectNumbers` and `generateNumbers` processes then appear. At some time later the corresponding finishing messages appear. The time the test took is then printed together with the result of the test signified by `OK` in this case.

```
.Collector Started
Numbers started
Collector Finished
Numbers finished

Time: 32.949

OK (1 test)
```

Output 17-1 Test Output

17.4 Summary

This chapter has shown that it is possible to test a system that is intended to run in parallel using an existing technology JUnit [] and GroovyTestCase formulation. The formulation described is somewhat limited in that only one test can be undertaken against the system under test, which is not the normal mode of operation within the JUnit framework.