6 Testing Parallel Systems: First Steps

Testing of systems is crucial and this chapter shows how GroovyTestCase can be used to test processes by

- identifying the process that is to be tested which must terminate
- creating support processes that terminate
- constructing the assertions that can be tested, using properties from the support processes

JUnit (JUnit, 2013) testing has become a widely accepted way of testing Java classes and there is a great deal of software support for this approach. In previous chapters, examples and exercises were introduced whereby the user had to ascertain for them self that the systems worked in the expected manner. This was achieved by looking at displayed output. This may be a satisfactory approach for small example systems but is not appropriate for systems that are to be used in an everyday context.

In this chapter the use of JUnit testing is introduced by using examples taken from earlier chapters. This will demonstrate that it is possible to use this approach and give a general architecture for testing parallel systems. The key to JUnit testing is that we test one or more assertions concerning the underlying implementation. In the parallel situation we have to identify a source of inputs that can be compared to the subsequent outputs for the assertion testing.

6.1 Testing Hello World

The testing of the `ProduceHW` and `ConsumeHello` processes (see Chapter 2) demonstrate that from the outset testing has to be considered at the time processes are designed and cannot be retrospectively added. To this end, properties are required that can be accessed once a process has terminated. These properties can then become components in any assertion. In this very simple case the `ProduceHW` process needs no alteration.

6.1.1 Revised ConsumeHelloForTest Process

The revised version of `ConsumeHelloForTest`, see Listing 6-1 requires the addition of a property `message` {13}, which is assigned {18} the values that have been read in from `inChannel` {16, 17}.

```
10 class ConsumeHelloForTest implements CSProcess {
11
12   def ChannelInput inChannel
13   def message
14
```

```
  void run() {
    def first = inChannel.read()
    def second = inChannel.read()
    message = "${first} ${second}!!!"
    println message
  }
}
```

Listing 6-1 The Revised Version of ConsumeHW

6.1.2 The HelloWorldTest Script

Listing 6-2 gives the script used to test ProduceHW and ConsumerHW.

The `ProduceHW` process from Chapter 2 is imported {10} into the testcase and thus provides a means of testing that process. The remainder of the coding is that required to build an instance of Groovy`TestCase`. This is the Groovy way of building JUnit tests. This requires the definition of a `void` method {14}, the name of which is prefixed with the word `test` that contains the script necessary to run the processes being tested. The primary requirement is that each of the processes **must** terminate. In many systems this is not feasible as the processes run in a loop that does not terminate. In Chapter 17 we shall see how to test such non-terminating process networks.

Alcatel·Lucent
www.alcatel-lucent.com/careers
What if you could build your future and create the future?
One generation's transformation is the next's status quo.
In the near future, people may soon think it's strange that devices ever had to be "plugged in." To obtain that status, there needs to be "The Shift".

Click on the ad to read more

```
10 import c02.ProduceHW
11
12 class HelloWorldTest extends GroovyTestCase {
13
14    void testMessage() {
15               def connect = Channel.one2one()
16               def producer = new ProduceHW ( outChannel: connect.out()
)
17               def consumer = new ConsumeHelloForTest ( inChannel:
                 connect.in() )
18
19               def processList = [ producer, consumer ]
20               new PAR (processList).run()
21               def expected = "Hello World!!!"
22               def actual = consumer.message
23               assertTrue(expected == actual)
24    }
25 }
```

Listing 6-2 The HelloWorldTest Script

The crucial elements are that we define each process as an instance {16, 17}. This is required so that we can access the `message` property of `ConsumeHelloForTest` when the system terminates. The processes are then run in parallel {19, 20}. The property `expected` is set to the `String` that should be output {21}. The `actual` value is obtained from the `message` property of `ConsumeHelloForTest` {22}. These are then compared {23} using the `assertTrue` method which produces an indication of whether the test passed.

6.2 Testing the Queue Process

The `Queue` Process discussed in Chapter 5.2 can be tested by sending a known number of test values into the `Queue` from the `QProducer` process and then ensuring that the same values are received by the `QConsumer` process. Listing 6-3 shows the modified `QProducerFortest` process. The only modifications required occur on {15}, where a new `List` property is added called `sequence`, which holds the sequence of produced values and on {23} where each produced value is appended (<<) to `sequence`. The printing of the produced values also has been removed. The `sequence` property is required to ensure we have a value that can be tested once the network of processes being tested has terminated.

```
10 class QProducerForTest implements CSProcess {
11
12   def ChannelOutput put
13   def int iterations = 100
14   def delay = 0
15   def sequence = []
16
17   void run () {
18     def timer = new CSTimer()
19
20     for ( i in 1 .. iterations ) {
21       put.write(i)
22       timer.sleep (delay)
23       sequence = sequence << i
24     }
25     put.write(null)
26   }
27 }
```

Listing 6-3 The Testable Version of QProducer Called QProducerForTest

Listing 6-4 shows the modified `QConsumerForTest` process, which as in the `QProducerFortest` defines a property that can be externally accessed, called `outSequence` {15}. The processing of the terminating `null` value has been modified {25} so that all the received values are appended to `outSequence` unless it is the `null` value, in which case the value of `running` is set `false`, causing the process to terminate.

```
10 class QConsumerForTest implements CSProcess {
11
12   def ChannelOutput get
13   def ChannelInput receive
14   def long delay = 0
15   def outSequence = []
16
17   void run () {
18     def timer = new CSTimer()
19     def running = true
20
21     while (running) {
22       get.write(1)
23       def v = receive.read()
24       timer.sleep (delay)
25       if ( v != null) outSequence = outSequence << v
26       else running = false
27     }
28   }
29 }
```

Listing 6-4 The Modified QConsumerForTest Process

6.3 The Queue Test Script

Listing 6-5 gives the GroovyTestcase class that causes `Queue` process testing. It takes the same basic structure as that used in the test of the Hello World system. It is important to note that the aim is to test the `Queue` process given in Chapter 5 and not the `QProducer` and `QConsumer` processes. The `Queue` process is imported {10}. The channels required to implement the network defined in Figure 5-2 are specified {14–16}. Instances of each of the processes are then defined {18–22}, so that we can subsequently test the values of the properties `sequence` and `outSequence` of `QProducer` and `QConsumer` respectively. The list of processes is then defined and executed in a `PAR` {23–24}, which must terminate if we are to be able to test the process properties in an assertion {28}.

```
10 import c05.Queue
11 class QueueTest extends GroovyTestCase {
12
13     void testQueue() {
14       def QP2Q = Channel.one2one()
15       def Q2QC = Channel.one2one()
16       def QC2Q = Channel.one2one()
17
18       def qProducer = new QProducerForTest ( put: QP2Q.out(), iterations: 50 )
19       def queue = new Queue ( put: QP2Q.in(), get: QC2Q.in(),
20       receive: Q2QC.out(), elements: 5)
21       def qConsumer = new QConsumerForTest ( get: QC2Q.out(),
22       receive: Q2QC.in() )
23       def testList = [ qProducer, queue, qConsumer ]
24       new PAR ( testList ).run()
25
26       def expected = qProducer.sequence
27       def actual = qConsumer.outSequence
28       assertTrue(expected == actual)
29     }
30 }
```

Listing 6-5 The QueueTest Script

The values of the `expected` and `actual` returned values are obtained from their processes and tested {26–28}. In more complex examples the construction of assertions is likely to be more elaborate depending upon the nature of the data being input and generated.

6.4 Summary

In this chapter we have introduced the concept of testing parallel systems, using the JUnit testing framework within a Groovy environment. The key requirement is that the network of processes must terminate. Further, the processes used to test the operation of the process network under test must contain properties that can be populated with data that can then be tested in one or more assertions. In Chapter 17 we reflect further on the testing of parallel systems and show how we can test systems that are designed not to terminate.

6.5 Exercises

Exercise 6-1

Construct a Test Case for the Three-To-Eight system constructed in the exercise for Chapter 2.