Communicating Process Architectures 2007
Alistair McEwan, Steve Schneider, Wilson Ifill and Peter Welch (Eds.)
IOS Press, 2007

Testing and Sampling Parallel Systems

Jon KERRIDGE
School of Computing, Napier University, Edinburgh, EH10 5DT

Abstract The testing of systems using tools such as JUnit is well known to the sequential programming community. It is perhaps less well known to the parallel computing community because it relies on systems terminating so that system outputs can be compared with expected outputs. A highly parallel architecture is described that allows the JUnit testing of non-terminating MIMD process based parallel systems. The architecture is then extended to permit the sampling of a continuously running system. It is shown that this can be achieved using a small number of additional components that can be easily modified to suit a particular sampling situation. The system architectures are presented using a Groovy implementation of the JCSP and JUnit packages.

Keywords: JUnit Testing, Sampling, GroovyTestCase, white-box, black-box

Introduction

The concept of testing, particularly using the *white-box* and *black-box* techniques, is well known and understood by the software engineering community. White-box testing is used to ensure that the methods associated with an object oriented class definition operate in the expected manner and that their internal coding is correct. Black-box testing is used to ensure that the overall operation of the class and its methods is as expected when operating in conjunction with other classes without concern for their internal coding.

The Agile programming community [1] has developed techniques commonly referred to as unit testing. In particular, these techniques have been incorporated into an open source framework that can be used with Java, called JUnit (www.junit.org). Typically, JUnit is used to undertake white-box testing. The use of the capability has been made even easier in the Groovy scripting environment by the creation of the `GroovyTestCase` [2], by for example, ensuring that all methods starting with `test` are compiled and executed as a Groovy script. A test will normally require some form of assertion test to check that an output value is within some bound or that some invariant of the system is maintained,

An ordinary object-oriented class uses its methods to pass messages between objects and thus need to be carefully tested. Hence the JUnit test framework has been designed specifically to undertake white-box testing of these methods. An object is tested by defining it as a *fixture*, which is then subjected to a sequence of tests. After each test is completed an assertion is evaluated to determine the success of the test. An assertion can test for a true or false outcome. The testing process requires the programmer to define an input sequence of calls to one or more methods of the object and also to specify the expected outcome. The assertion tests the generated output from the object under test against the expected outcome. Thus programming becomes a process of defining inputs and expected outputs and writing the program to achieve the desired outputs. The JUnit framework automates this process further by combining sequences of tests into *testsuites*. If a change has been made to the underlying object all the tests contained in all the testsuites can be run to ensure the change has not created any unwanted side effects.

In the MIMD parallel processing environment, using JCSP (www.jcsp.org), the classes implement the interface `CSProcess` that has only one method `run()`. Any methods are private to the object and used simply to assist in the coding of the process. Hence the use of unit testing in the parallel environment can be considered more akin to black-box testing because there is only one method to test. Often processes are written in a style that runs forever rather than having a specific termination strategy. Processes can be returned to a known state using techniques such as poison [3, 4] but unless specifically required tend not to be used. Even in this situation, the process can still continue to run and may not terminate. If a network of processes does terminate then the normal testcase framework can be used. Hence a means of testing a non-terminating system has to be specially designed so the non-terminating part under test can continue to run, while the testing part terminates so that data values can be extracted for assertion testing. If the network of processes does not terminate we can never extract the values from its execution that are required to test the associated assertions. If the system has been designed to run forever then the addition of code to cause the system to terminate means the system being tested is not the one that will be used in any final deliverable. We therefore need to create a bi-partite test environment in which the process network under test is able to run forever. A terminating test part injects a finite sequence of values, with an expected outcome into the network under test. The test part also receives outputs which can be assertion tested against the expected outcome. This simple strategy is impossible to run as a single process network in a single processing node because even though the processes in the test part will terminate the network under test will not terminate and thus the complete network never terminates and thus the assertions cannot be tested. The use of the `GroovyTestCase` framework means that the testing can be even more easily automated.

Sampling a system provides a means of checking that a system remains within pre-defined bounds as it operates normally. The benefit of providing a sampling architecture that is different from the testing architecture is that it can be incorporated into the system either at design time or once it has been implemented. The primary requirement is that the processes that are used to extract the samples are as lightweight as possible. Crucially, these sampling processes must not result in any modification to the system that has already been tested.

In the next section, a generic testing architecture is presented that utilizes the capability of JCSP to place processes on different nodes of a multi-node processing network connected by means of a TCP/IP network. Section 2 then demonstrates how this architecture can be applied to a teaching example process network. Section 3 then shows how the same process network could be sampled during normal operation. Sections 4 and 5 then describe generic sampling architectures for systems that respectively communicate data by means of object data transfers and by primitive data types. Finally, some conclusions are drawn and further work identified.

1. A Generic Testing Architecture

Figure 1 shows a generic architecture in which it is presumed that the Process-Network-Under-Test (PNUT) is either a single process or a collection of processes that does not terminate. The Input-Generator process produces a finite set of inputs to the PNUT and may also create a data structure that can form one part of a test assertion. Similarly, the Output-Gatherer process collects data from the PNUT and stores it in a data structure that can be subsequently tested by Test-Network. The Assertion-Testing is only undertaken when both the Input-Generator and Output-Gatherer processes have terminated.

The goal of the architecture is to create a means by which each of the processes or subnetworks of processes can be tested and shown to operate according to the tests that have been defined for that particular process or sub-network of processes. The JCSP, due to its reliance on CSP provides a compositional semantics when processes are combined into larger networks. Other mechanisms are available, such as FDR [5] for determining the deadlock freedom of such compositions, but cannot then test the full range of values that might be applied to such a network and thus the need for a testing framework for parallel systems. The PNUT shown in Figure 1 might be a single process or a network of processes that together form a collection of testable processes that are subsequently used in a compositional manner in the final system design.

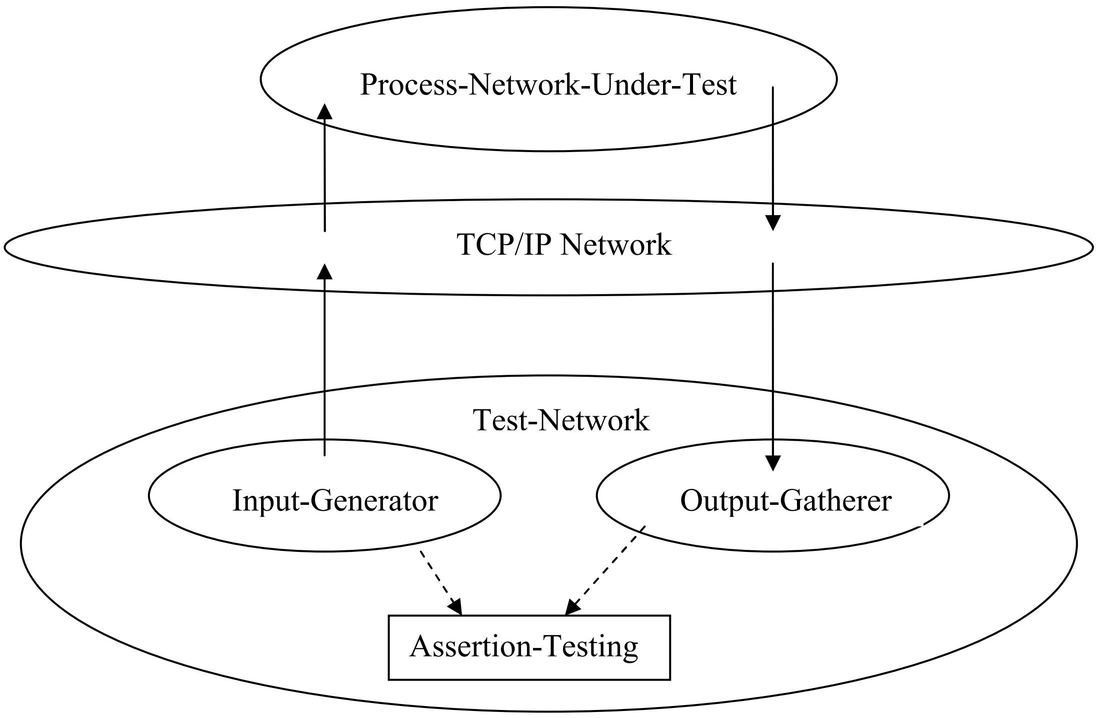

Figure 1 Generic Testing Architecture

Both the Input-Generator and Output-Gatherer processes must run as a Parallel within the process Test-Network, then terminate after which their internal data structures can be tested within Assertion-Testing. An implementation of the Test-Network process for a specific example is shown in Listing 1. It does however demonstrate the generic nature of the architecture in that the only part that has to be specifically written are the `GenerateNumbers` and `CollectNumbers` processes that implement the Input-Generator and Output-Gatherer respectively.

The class RunTestPart implements the Test-Network {1}[1]and simply extends the class `GroovyTestCase`. The method `testSomething` {3} 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 {5}. Two `NetChannels`, `ordinaryInput` {7}and `scaledOutput` {8} 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 {10, 11} using the techniques described in [6] using

[1] The notation {n} indicates a line number in a lisitng.

Groovy parallel helper classes. The processes are then invoked {13, 15}. Once the `PAR` has terminated, the properties `generatedList`, `collectedList` and `scaledList` can be obtained from the processes {17-20} 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 {21}. 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 {22}.

```
01  class RunTestPart extends GroovyTestCase {
02
03    void testSomething() {
04
05        Node.getInstance().init(new TCPIPNodeFactory ())
06
07        NetChannelOutput ordinaryInput = CNS.createOne2Net("ordinaryInput")
08        NetChannelInput scaledOutput = CNS.createNet2One("scaledOutput")
09
10        def collector = new CollectNumbers ( inChannel: scaledOutput)
11        def generator = new GenerateNumbers (outChannel: ordinaryInput)
12
13        def testList = [ collector, generator]
14
15        new PAR(testList).run()
16
17        def original = generator.generatedList
18        def unscaled = collector.collectedList
19        def scaled = collector.scaledList
20
21        assertTrue (original == unscaled)
22        assertTrue (TestUtilities.list1GEList2(scaled, original))
23
24    }
25
26  }
```

Listing 1 An Implementation of the Test-Network Process

The benefit of this approach is that we are guaranteed that the Test-Network will terminate and thus values can be tested in assertions. The fact that the PNUT continues running is made disjoint by the use of the 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.

1.1 Example Generator and Gatherer Processes

Necessarily, the Generator and Gatherer processes will depend upon the PNUT. Listing 2 shows a typical formulation of a Generator process, which produces a finite sequence of numbers. The properties of the process will vary, however `outChannel` and `generatedList` will always be required. The channel is used to communicate values to the PNUT {38} and `generatedList` provides a means of storing the output sequence in a property {33, 39} that can be accesses once the process has terminated. The operator << {39} appends a value to a list. In this case the numbers need to be output with a `delay` {29, 36, 40} of one second between each number. The size of the output sequence can easily be altered by varying the value assigned to the property `iterations`, which has a default value of 20.

```
class GenerateNumbers implements CSProcess{

  def delay = 1000
  def iterations = 20

  def ChannelOutput outChannel
  def generatedList = []

  void run() {
    def timer = new CSTimer()
    for (i in 1 .. iterations) {
      outChannel.write(i)
      generatedList << i
      timer.sleep(delay)
    }
  }
}
```

Listing 2 An Implementation of a Generator Process

Listing 3 similarly gives the code for a Gatherer process. In this case two output lists can be collected, one which is the same as the original data stream (`collectedList`) and one which has been modified in some manner (`scaledList`). These lists will be accessible when the Test-Network process terminates as they are properties of the `CollectNumbers` process. The `result`s are `read` from `inChannel` as objects of type `ScaledData`, whose properties `original` and `scaled` are appended to each of the accessible property lists.

```
class CollectNumbers implements CSProcess {

  def ChannelInput inChannel
  def collectedList = []
  def scaledList = []

  def iterations = 20

  void run() {
    for ( i in 1 .. iterations) {
      def result = (ScaledData) inChannel.read()
      collectedList << result.original
      scaledList << result.scaled
    }
  }
}
```

Listing 3 A Gatherer Process

The basic structure of the Test-Network processes is essentially independent of the PNUT, though it does need to be specialized to its specific requirements with respect to the number and type of input channels and of its outputs. The key requirement is that some relationship between the inputs and outputs has to be testable in an assertion.

2. The Network Under Test

The Network-Under-Test used in the above example is based upon the scaling device described by Belapurkar [7]. The scaling device reads integers that appear on its input every second, hence the delay introduced in the `GenerateNumbers` process shown in Listing 2. The scaling device then outputs these inputs multiplied by a constant factor, which is initially set to 2. The constant factor is however doubled every 5 seconds. Additionally, a controlling mechanism is provided that suspends the normal operation of the scaling device. The current value of the scaling factor is read and modified as necessary. In this case, the scaling factor is incremented by 1. While the scaling device is in the suspended state any input value is output without any scaling. The scaling device is suspended after 7 seconds and remains in the suspended state for 0.7 seconds. The script that runs the scaling device is given in Listing 4.

```
60  Node.getInstance().init(new TCPIPNodeFactory ())
61
62  NetChannelInput ordinaryInput = CNS.createNet2One("ordinaryInput")
63  NetChannelOutput scaledOutput = CNS.createOne2Net("scaledOutput")
64
65  new PAR(new ScalingDevice (inChannel: ordinaryInput,
66                             outChannel: scaledOutput) ).run()
```

Listing 4 Script to Execute the ScalingDevice in its own JVM

A network node is created {60} followed by two net channels that are the ones corresponding to those created within the Test_Network {7, 8}. The single `ScalingDevice` process is then executed {65, 66}. This process does not terminate.

The `ScalingDevice` is defined by the process shown in Listing 5, which defines two channel properties for input to and output from the process {69, 70}. The `run()` method uses three channels to connect the `Scale` process to the `Controller` process. These are used to implement the suspend, reading and updating of the scale factor described above.

```
67  class ScalingDevice implements CSProcess {
68
69    def ChannelInput inChannel
70    def ChannelOutput outChannel
71
72    void run() {
73      def oldScale = Channel.createOne2One()
74      def newScale = Channel.createOne2One()
75      def pause = Channel.createOne2One()
76
77      def scaler = new Scale ( inChannel: inChannel,
78                               outChannel: outChannel,
79                               factor: oldScale.out(),
80                               suspend: pause.in(),
81                               injector: newScale.in(),
82                               scaling: 2 )
83
84      def control =  new Controller ( testInterval: 7000,
85                                      computeInterval: 700,
86                                      factor: oldScale.in(),
87                                      suspend: pause.out(),
88                                      injector: newScale.out() )
89
90      def testList = [ scaler, control]
91
92      new PAR(testList).run()
93    }
94
95  }
```

Listing 5 The Definition of the `ScalingDevice` Process

The unit test described above demonstrates that the basic functionality of the scaling device is correct. However, can we be assured that the system will behave correctly over a longer period? A moment's reflection will indicate this is not the case because if you continue to double the scaling factor and add one every so often then the bound on integer values will be reached and overflow will occur. Given that we know this to be the case can we demonstrate it by means of a generalised sampling environment that is applicable in a wide variety of situations? Further, could such a sampling system be left in place permanently so that the operation of the system can be checked periodically?

3. Sampling the Scaling Device

We shall use exactly the same Scale and Controller processes as those used in the unit testing; otherwise there was no point in testing them! We shall however drive them in a slightly different manner so that we can run the system for an indeterminate period. This is shown in Figure 2.

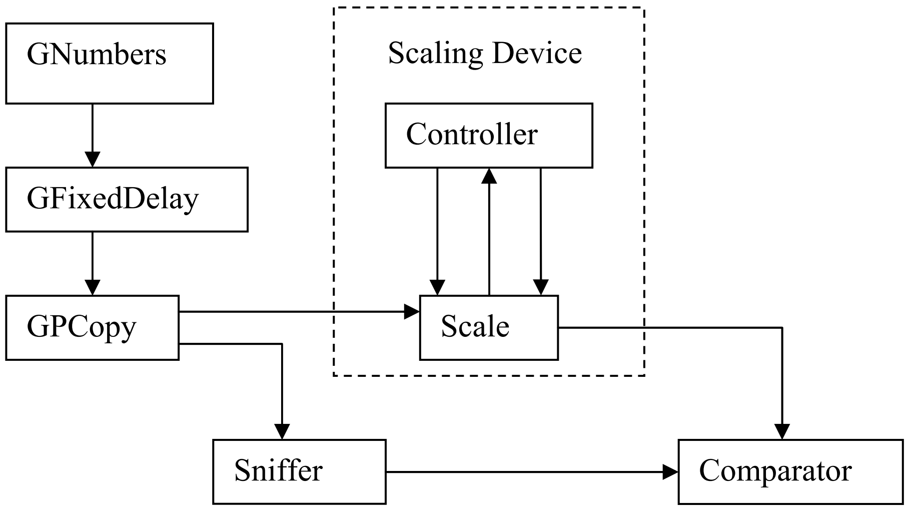

Figure 2 Sampling the Scaling Device

GNumbers, GFixedDelay are Groovy versions of the equivalent Numbers and FixedDelay processes of the package jcsp.plugNplay. GPCopy achieves the same effect as Delta2 in the same package. GNumbers generates a sequence of integers. GFixedDelay introduces a delay into the communication, in this case of 1 second. GPCopy copies any input to both its outputs in parallel.

The Sniffer process inputs all the data output to it by GPCopy. If a predefined time has elapsed since the data stream was last sniffed then the next input from GPCopy is output to the Comparator process. The Comparator process reads all the outputs from the Scale process. The Scale output comprises a data object containing both the original unscaled value and the scaled value. The definition of the scaling device is such that we know that the scaled value should either be greater than or equal to the original value. The Comparator already knows the sniffed original value and can thus check the relationship between the original value and the scaled value is correct, when it reads a record with the sniffed original value. This does presume that it takes longer for the Scale process to operate than the communication between the Sniffer and the Comparator.

On running this system we soon discover that the expected failure does occur and the scaled value goes negative and thus out of bounds. Perhaps more surprisingly a different failure mode is observed if the system is allowed to run further. We discover that the system goes into an infinite loop of a scaling factor sequence of -2, -1, -2, -1, …when we might have expected it to return to a positive number sequence as the values overflow a second time. Recall (Section 2) that the operations undertaken on the scaling factor are one of doubling and then adding one and thus this outcome is entirely reasonable but hard to predict when the processes are being defined because the operations are undertaken in different processes. The same outcome would be achieved if we quadrupled and then added three, except that the sequence would be -1, -4, -1, -4, … .

Listing 6 shows the coding of the `Sniffer` process. The process has two channel properties, one, `fromSystemCopy`, reads the outputs from the `GPCopy` process and the other, `toComparator`, writes data to the `Comparator` process, see Listing 7. The final property is the `sampleInterval`, which defaults to 10 seconds.

```
96   class Sniffer implements CSProcess{
97
98     def ChannelInput fromSystemCopy
99     def ChannelOutput toComparator
100    def sampleInterval = 10000
101
102    void run() {
103      def TIME = 0
104      def INPUT = 1
105      def timer = new CSTimer()
106      def snifferAlt = new ALT([timer, fromSystemCopy])
107      def timeout = timer.read() + sampleInterval
108      timer.setAlarm(timeout)
109      while (true) {
110        def index = snifferAlt.select()
111        switch (index) {
112          case TIME:
113            toComparator.write(fromSystemCopy.read())
114            timeout = timer.read() + sampleInterval
115            timer.setAlarm(timeout)
116            break
117          case INPUT:
118            fromSystemCopy.read()
119            break
120        }
121      }
122    }
123  }
```

Listing 6 The Sniffer Process Code

The `run` method defines a `CSTimer` {105}that is used to generate an alarm when the `sampleInterval` has elapsed {107, 114}. During normal `INPUT` {118} the data from the channel `fromSystemCopy` is read and ignored. When the alarm `TIME` has occurred the next value `fromSystemCopy` is read {113}and written to the channel `toComparator`. The next alarm time is recalculated {114,115}.

The `Comparator` process receives outputs from the system being sampled as well as inputs from the `Sniffer` process {126, 127}. The `Comparator` alternates over these inputs {132}. On receipt of a value from the `Sniffer` {137}, the process reads values from the system until the value to be evaluated is input {140-150}. It then tests the value to determine its relationship to an invariant of the system. An appropriate message is printed, which in a real system could be stored in a database.

The `Sniffer` and `Comparator` have been implemented knowing the detailed operation of the Scaling Device; in particular that it inputs a stream of integers and outputs objects containing both the original and the modified value. What happens if the input and output stream are either both objects or both streams of base types such as int, float etc?

Such requirements cannot be easily combined into a single architecture and thus in the following sections we describe approaches that enable sampling of the two types of system. Inevitably, though, the ability to create a generic sampling architecture similar to the generic testing architecture, described in Section 2, will be somewhat more difficult as the nature of the sampling system will vary with the specifics of the systems being sampled.

```
class Comparator implements CSProcess {

  def ChannelInput fromSystemOutput
  def ChannelInput fromSniffer

  void run() {
    def SNIFF = 0
    def COMPARE = 1
    def comparatorAlt = new ALT ([fromSniffer, fromSystemOutput ])
    def running = true
    while (running) {
      def index = comparatorAlt.priSelect()
      switch (index) {
        case SNIFF:
          def value = fromSniffer.read()
          def comparing = true
          while (comparing) {
            def result = (ScaledData) fromSystemOutput.read()
            if (result.original == value){
              if (result.scaled >= result.original) {
                println "Within bounds: ${result}"
                comparing = false
              }
              else {
                println "Outwith Bounds: ${result}"
                running = false
              }
            }
          }
          break
        case COMPARE:
          fromSystemOutput.read()
          break
      }
    }
  }
}
```

Listing 7 The Comparator Process

4. An Object Based Sampling System

The basis of this sampling architecture relies on the ability of object oriented systems to extend a class such that any process that is unaware of the extension will be unable to manipulate the extended object definition. The generic architecture is shown in Figure 3.

The DataGenerator process represents a source of input objects to the Sampled Network. The Sampler process copies all inputs to its output unchanged unless it has received an input from the SamplingTimer process. The SamplingTimer process generates an output at predefined time intervals, known as the sampling period. In normal operation the Sampler process will just output the object generated by the DataGenerator. After receiving and input from the SamplingTimer the Sampler process will output an extended version of the object.

This extended version of the data object will have no effect on the SampledNetwork because it only recognizes the non-extended object. All outputs will be processed by the Gatherer process. All outputs from the Gatherer process are output to a subsequent part of the system, which in this case is a GPrint process. GPrint causes the printing of any object, provided it has a `toString()` method. Any extended data object will, in addition be communicated to the Evaluator process where its content can be evaluated against the invariants of the SampledNetwork.

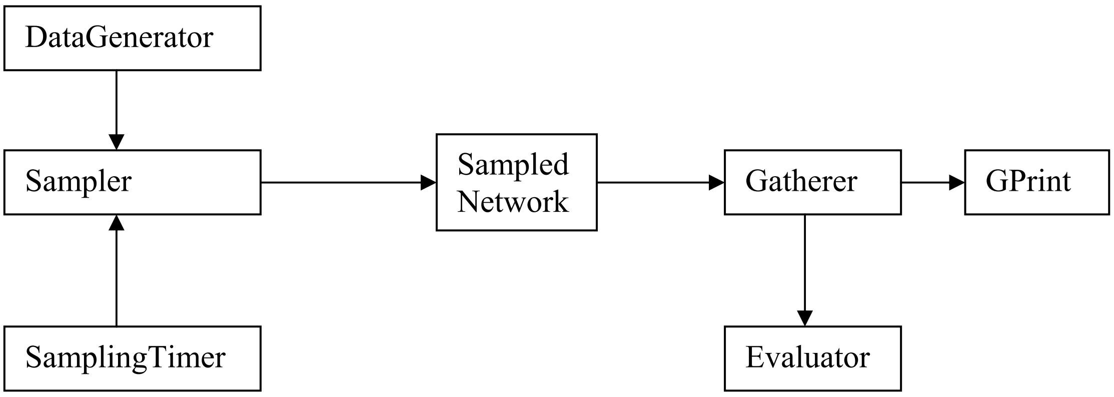

Figure 3 Generic Sampling Architecture for Object Based Input and Output

Listing 8 shows the code of the `SamplingTimer` process, which employs a simple loop that sleeps for the `sampleInterval` {169} and then outputs a signal message {170} on its `sampleRequest` channel

```
161 class SamplingTimer implements CSProcess {
162
163   def ChannelOutput sampleRequest
164   def sampleInterval
165
166   void run() {
167     def timer = new CSTimer()
168     while (true){
169       timer.sleep(sampleInterval)
170       sampleRequest.write(1)
171     }
172   }
173 }
```

Listing 8 The SamplingTimer Process

The `Sampler` process is shown at Listing 9.The channel `inChannel` receives inputs from the DataGenerator, which are output on the `outChannel` {191}. The `sampleRequest` channel is used to input requests for the generation of a sample. On receipt of the request signal {185}, the next data input is read {186} and its data values are used to create an instance of the extended object, referred to as `FlaggedSystemData` because the extension is simply a Boolean value set `true`. This extended object is then written to the system.

Necessarily, the `Evaluator` process is system dependent, an example of which is shown in Listing 10. Quite simply, values from the extended object are tested against each other {204} and a suitable message printed or saved to a database.

```
174 class Sampler implements CSProcess {
175
176   def ChannelInput inChannel
177   def ChannelOutput outChannel
178   def ChannelInput sampleRequest
179
180   void run() {
181     def sampleAlt = new ALT ([sampleRequest, inChannel])
182     while (true){
183       def index = sampleAlt.priSelect()
184       if (index == 0) {
185         sampleRequest.read()
186         def v = inChannel.read()
187         def fv = new FlaggedSystemData ( a: v.a, b:v.b, testFlag: true)
188         outChannel.write(fv)
189       }
190       else {
191         outChannel.write(inChannel.read())
192       }
193     }
194   }
195 }
```

Listing 9 The Sampler Process

```
196 class Evaluator implements CSProcess {
197
198   def ChannelInput inChannel
199
200   void run() {
201     while (true) {
202       def v = inChannel.read()
203       def ok = (v.c == (v.a +v.b))
204       println "Evaluation: ${ok} from " + v.toString()
205     }
206   }
207 }
```

Listing 10 The Evaluator Process

The `Gather` process, shown in Listing 11, repeatedly reads in objects from its `inChannel` and determines the type of the input {216, 217}.

```
208 class Gatherer implements CSProcess {
209
210   def ChannelInput inChannel
211   def ChannelOutput outChannel
212   def ChannelOutput gatheredData
213
214   void run(){
215     while (true){
216       def v = inChannel.read()
217       if ( v instanceof  FlaggedSystemData) {
218         def  s = new SystemData ( a: v.a, b: v.b, c: v.c)
219         outChannel.write(s)
220         gatheredData.write(v)
221       }
222       else {
223         outChannel.write(v)
224       }
225     }
226   }
227 }
```

Listing 11 The Gatherer Process

If the object has been extended then a non-extended version of the data is constructed and output to the rest of the system {218, 219}. The extended version of the data is also written to the Evaluator process {220}. Normally, the input data is just written to the output channel {223}.

Typical output from the sampling system is shown below where we see that the flagged data values are output twice once from `GPrint` as non-extended data and once from the `Evaluator` process where the complete `FlaggedSystemData` is printed.

```
System Data: [58, 59, 117]
System Data: [60, 61, 121]
Evaluation: true from Flagged System Data: [60, 61, 121, true]
System Data: [62, 63, 125]
System Data: [64, 65, 129]
System Data: [66, 67, 133]
System Data: [68, 69, 137]
System Data: [70, 71, 141]
System Data: [72, 73, 145]
System Data: [74, 75, 149]
System Data: [76, 77, 153]
System Data: [78, 79, 157]
System Data: [80, 81, 161]
Evaluation: true from Flagged System Data: [80, 81, 161, true]
System Data: [82, 83, 165]
System Data: [84, 85, 169]
```

5. Sampling Systems That Do Not Use Data Objects Explicitly

For systems that do not use objects explicitly, we could count the inputs to the Sampled Network. A suitable architecture is shown in Figure 4.

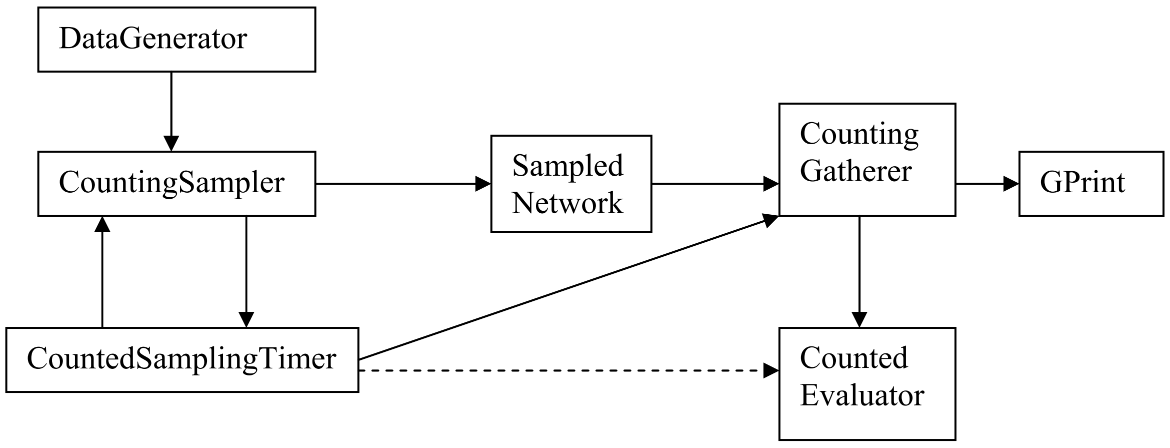

Figure 4 Generic Architecture for Sampling Networks by Counting Inputs

The CountingSampler takes inputs from the DataGenerator and copies them to the Sample Network, keeping a count of each input. At a rate determined by the sampling interval the CountedSamplingProcess will make a request to the CountingSampler, which will respond with the count value of the data input to be sampled. The CountedSamplingTimer will receive the value of the count which it sends to the CountingGatherer process. The CountingGatherer process keeps a count of the outputs from the Sample Network and on receipt of the output that corresponds to the count value it has received it outputs the count value and the output value to the CountedEvaluator process. The CountedEvaluator process can then record the sampled value and any result from a test that has been carried out. In some cases, the processes could be modified so that the CountingSampler process returns more than just the count value, for example the input data value, to the CountedSamplingTimer; in which case, this additional data may be communicated to the CountedEvaluator. The only requirement is that the time taken to undertake the two communications from the CountingSampler via the CountedSamplingTimer to the CountingGatherer must be less than the time taken to process the data in the Sampled Network. In this simple implementation we also need to ensure that every input to the Sampled Network has a corresponding output.

As a general comment we note that there is a large similarity between the architecture shown in Figures 3 and 4. This leads to the observation that some form of generic framework could be constructed that permits the easier construction of sampling architectures much in the same way as the `GroovyTestCase` framework has simplified the already relatively easy JUnit Test Case Framework.

6. Conclusions and Further Work

The paper demonstrates that is possible to use standard testing techniques commonly adopted by the software engineering community to the specialized requirements of parallel systems testing. In particular, a technique has been demonstrated that enables black-box testing using the GroovyTestCase specialization of JUnit. The paper then demonstrated that a simple set of additional processes could be easily defined that permit the sampling of running systems using a variety of approaches depending upon the nature of the data transmitted around the system.

The primary area for further work is to take the basic sampling processes and form them into a framework so they can be more easily incorporated by designers of parallel systems into their designs. In particular, the use of Groovy builders will hopefully make this a much simpler task than might be expected.

The nature of the processes being tested in this paper is somewhat limited because they were all deterministic in nature because the expected outcome was always fully determined by the input sequence. The architecture needs to be further developed to cope with non-deterministic systems where the final output is not fully determined by the input. To a certain extent the scaling system was non-deterministic in that the operation of the `Control` process was asynchronous with the `Scale` process. However, for means of explanation, it was made fully determined. Even for completely non-deterministic systems, there will probably be some processes or collection of processes that can be tested in a deterministic manner. The sampling of non-deterministic process collections will be easier to construct as the nature of the specific sampling processes is more closely tied to the underlying system.

Acknowledgements

John Savage introduced me to the delights of the GroovyTestCase specialization of the JUnit framework and convinced me that such testing could be achieved in parallel systems. He then convinced me to investigate simple ways of sampling systems. Kevin Chalmers suggested the technique of extending an object because he is a far better object oriented programmer than I. The comments of the referees were invaluable in improving the presentation of the material in this paper.

References

[1] K. Beck, Test Driven Development: By Example, Addison-Wesley, ISBN-10: 0-321-14653-0, 2003
[2] K. Barclay and J. Savage, Groovy Programming: An Introduction for Java developers, Morgan Kaufmann, San Fransisco, CA, ISBN-10: 0-12-372507-0, 2007
[3] PH Welch, Graceful Termination – graceful resetting, in A Bakkers (ed), Proceedings OUG-10: Applying Transputer Based parallel Machines, IOS Press, pp310-317, 1989.
[4] BHC Sputh, and AR Allen, JCSP-Poison: Safe Termination of CSP Process Networks, in J Broenink et al (eds), Communicating Process Architectures, 2005, IOS Press, pp 71-107, 2005
[5] FDR2 User Manual, Formal Systems Europe Ltd, http://www.fsel.com/fdr2_manual.html , accessed 11th April 2007.
[6] J.Kerridge, K. Barclay and J. Savage, Groovy Parallel: A Return to the Spirit of occam?, Proceedings of Communicating Process Architectures 2005, IOSPress, Amsterdam, 2005
[7] Belapurkar A, http://www-128.ibm.com/developerworks/java/library/j-csp2/ accessed 11th April 2007.