5 Extending the Alternative: A Scaling Device and Queues

Many machines used in automated processes have some means of monitoring their operation, for example, by calculating running averages of specific values and ensuring they stay within a specified range. If they go out of range then the machine recalibrates itself. In this chapter we shall build a model of such a device, but without having to interface to a real machine!

5.1 The Scaling Device Definition

The scaling device[1] reads incoming integers that arrive every second. The device then multiplies the incoming value by its current scaling factor, which it then outputs, together with the original value. The scaling factor is doubled at a regular interval, of say, 5 seconds. In addition, there is a controlling function that suspends the operation of the scaling device again at regular intervals, of say, 7 seconds to simulate the testing of its operation. When it is suspended the scaling device outputs its current scaling factor to the controller. At some time later, the controller, having computed another scaling factor, will inject the new scaling factor into the controller, which resumes its normal mode of operation. While the scaling device is suspended by the controller it outputs all input values unscaled.

The structure of the system, showing the channels that will be used for the communications specified above is shown in Figure 5-1.

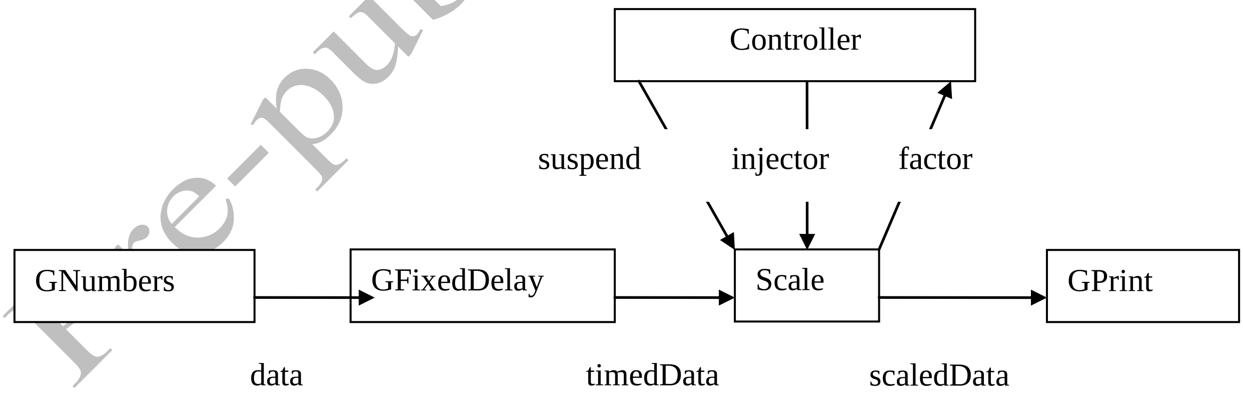

Figure 5-1 Structure of the Scaling Device

The processes `GNumbers`, `GFixedDelay` and `GPrint` are available in the package `groovyPlugAndPlay`. Thus the discussion revolves around the structure of the remaining two processes.

---

[1] Belapurkar A, http://www-128.ibm.com/developerworks/java/library/j-csp2/

5.1.1 The Controller Process

The code that implements the `Controller` process is shown in Listing 6-1.

```
01      class Controller implements CSProcess {

02        def long testInterval = 7000
03        def long computeInterval = 700
04        Def int addition = 1

05        def ChannelInput factor
06        def ChannelOutput suspend
07        def ChannelOutput injector

08        void run() {
09          def currentFactor = 0
10          def timer = new CSTimer()
11          def timeout = timer.read()

12          while (true) {
13            timeout = timeout + testInterval
14            timer.after ( timeout )
15            suspend.write (0)
16            currentFactor = factor.read()
17            currentFactor = currentFactor + addition
18            timer.sleep(computeInterval)
19            injector.write ( currentFactor )
20          }
21        }
22      }
```

Listing 5-1 Code of the Controller Process

From Figure 5-1 we can see that `Controller` has three channel properties {4-6}. In addition, it has two timeout values, one `testInterval` {2} determines the period between successive tests of the scaling device. The other, `computeInterval` {3} is used to simulate the time it takes to compute the revised scaling factor.

The JCSP class `CSTimer` provides a means of manipulating time in a consistent and coherent manner. An instance of `CSTimer`, called `timer` is defined {10}. The `timer` can be read at any instant and the current `long` value of the system clock in milliseconds is returned, which also justifies the type `long` for the interval properties defined previously. The value of `timeout` is set to the current time {11}. The device operates as a never ending loop {12-21}, which for most machine tools is reasonable.

Within the loop the timeout is incremented by the `testInterval` {13}, which must be some time in the future. The `after` operation on `timer` causes the process to be suspended until the value of the current time is after the indicated alarm time. While a process is suspended in this manner it will consume no processing resource. Once the `testInterval` has elapsed, the `Controller` writes a signal to the `Scale` process to suspend its operation {15}. The value communicated does not matter, so the value 0 is perfectly adequate. The `Controller` then reads the current scaling factor from the `Scale` process into `currentFactor` using the channel `factor` {16}. The value of `currentFactor` is then incremented {17} by the value contained in the property `addition` {4}, to simulate a change in the scaling factor. The time to undertake this recalculation is then simulated by suspending the process for the `computeInterval` by calling the `sleep` method on the `timer` {18}. The `sleep` method deschedules the process for the specified sleeping time. The process consumes no processor resource while it is sleeping. In this case the effect of `after` and `sleep` are the same, achieved in a different manner. In some situations, the `after` method will be the more appropriate because it provides relative time. The `sleep` method provides an absolute value. Once the process has been rescheduled, it writes {19} the newly computed `currentFactor` on the `injector` channel to the `Scale` process.

5.1.2 The Scale Process

The structure of the `Scale` process is shown in Listings 5-2 and 5-3. The operation of the `Scale` process can be partitioned into two distinct parts; when it is operating in the normal mode and when it is suspended. In the normal mode it will accept inputs from the channels `timedData` and `suspend`, see Figure 5-1. It will also respond to timer alarms indicating that the scaling factor should be doubled. In the suspended mode it will only respond to inputs from the channels `timedData` and `injector`. To reflect these situations a set of guards will be needed for each mode. Furthermore, the suspended set will only be considered when the process has moved from the normal mode into the suspended mode.

```
23     class Scale implements CSProcess {

24       def int scaling = 2
25       Def int multiplier = 2
26       def ChannelOutput outChannel
27       def ChannelOutput factor
28       def ChannelInput inChannel
29       def ChannelInput suspend
30       def ChannelInput injector

31       void run () {
32         def SECOND = 1000
33         def DOUBLE_INTERVAL = 5 * SECOND
34         def NORMAL_SUSPEND  = 0
35         def NORMAL_TIMER    = 1
36         def NORMAL_IN       = 2
37         def SUSPENDED_INJECT = 0
38         def SUSPENDED_IN     = 1

39         def timer = new CSTimer()
40         def normalAlt = new ALT ( [ suspend, timer, inChannel ] )
41         def suspendedAlt = new ALT ( [ injector, inChannel ] )

42         def timeout = timer.read() + DOUBLE_INTERVAL
43         timer.setAlarm ( timeout )
```

Listing 5-2 The Properties and Initialisation of the Scale Process

The channel properties are defined {24-30}, together with the initial `scaling` value {24} and the `multiplier` that will be applied to the scaling factor {25}. The `inChannel` property {28} is connected to `timedData` of Figure 5-1 and `outChannel` to `scaledData` {26}. Within the `run()` method a number of constants are defined; `DOUBLE_INTERVAL` {33} specifies the number of milliseconds between the doubling of the `scaling` factor. The remainder are constants {34-38} used to identify which `case` is to be considered when the `switch` statements associated with the alternatives are processed. A `timer` is defined {39}, followed by the two different alternatives {40, 41}. Both of the alternatives will be accessed using a `priSelect` method and thus the ordering of the guards in the alternatives is important and should always start with the highest priority going to the lowest in sequence. The alternative `normalAlt` applies when the device is not in a `suspended` state. The highest priority guard is that associated with the `suspend` channel. The next highest will result from a `timer` alarm and the lowest is the input of some data on the `inChannel`. In the `suspended` state the `suspendedAlt` will apply and this is just an alternation over the `injector` and `inChannel` channels because `timer` alarms are ignored. At {42} the `timeout` for the first doubling of the scaling factor is defined by reading the `timer` and adding the doubling interval. An alarm on the `timer` is made by calling the method `setAlarm` {43} with the required time, which must be some time in the future. This means that `normalAlt` will be enabled on the `timer` alternative once the value of the `timer` has increased beyond `timeout`. A timer contained within an alternative guard that is disabled, consumes no processor resource, until the alarm is enabled.

```
44          while (true) {
45            switch ( normalAlt.priSelect() ) {

46              case NORMAL_SUSPEND :
47                suspend.read()
48                factor.write(scaling)
49                def suspended = true
50                println "Suspended"

51                while ( suspended ) {
52                  switch ( suspendedAlt.priSelect() ) {

53                    case SUSPENDED_INJECT:
54                      scaling = injector.read()
55                      println "Injected scaling is ${scaling}"
56                      suspended = false
57                      timeout = timer.read() + DOUBLE_INTERVAL
58                      timer.setAlarm ( timeout )
59                      Break

60                    case SUSPENDED_IN:
61                      def inValue = inChannel.read()
62                      def result = new ScaledData()
63                      result.original = inValue
64                      result.scaled = inValue
65                      outChannel.write ( result )
66                      break
67                  }  // end-switch
68                } //end-while
69                break

70              case NORMAL_TIMER:
71                timeout = timer.read() + DOUBLE_INTERVAL
72                timer.setAlarm ( timeout )
73                scaling = scaling * multiplier
74                println "Normal Timer: new scaling is ${scaling}"
75                break

76              case NORMAL_IN:
77                def inValue = inChannel.read()
78                def result = new ScaledData()
79                result.original = inValue
80                result.scaled = inValue * scaling
81                outChannel.write ( result )
82                break
83            } //end-switch
84          } //end-while
85        } //end-run
86      }
```

Listing 5-3 The Scale Process Main Loop

The main loop of the device Listing 5-3, comprises {44-84} and is created by means of a never ending `while` loop. At the start of the main loop the device is presumed to be in the normal state and thus we `switch` on the `normalAlt` {45}. If none of the guards is ready the process waits until one becomes enabled. Each time an alternative is executed the guards are evaluated to determine which are enabled and then a selection is made form the ready ones according to the type of `select` operation undertaken.

If the enabled alternative results from an input on the `suspend` channel then the `case NORMAL_SUSPEND` will be obeyed {46}. First, the channel `suspend` must be `read` {47}, the value of which can be ignored because this is just a signal to enter the suspend state. Recall that the `Controller` process wrote a nominal value {15} of 0. The `Scale` process then writes its current `scaling` factor to the `factor` channel {48}. The property `suspended` is defined and set `true` {49}. A message is printed {50} and then the loop associated with the `suspended` state is entered {51}. In this state the process `switches` on `suspendedAlt` {52}, which has two alternatives.

If the enabled alternative is an input on the `injector` channel the case `SUSPENDED_INJECT` is obeyed {53}. The new value of `scaling` is `read` from the `injector` channel {54} and a message displaying the new factor printed {55}. The value of `suspended` is now reset {56} to `false`, which will cause the controlling while loop {51} to terminate. Because the `injector` input is also taken as an indication that normal operation can resume, the `timer` alarm can be reset {57-58}.

In the `suspended` state, the only other alternative that can occur, results from input on the `inChannel`, this causes the `SUSPENED_IN` case to be obeyed {60}. The channel `inChannel` is `read` into `inValue` {61}. A variable `result` of type `ScaledData` is defined {62}, see Listing 5-4. The device in the `suspended` state does not apply the `scaling` to any incoming data and so both the `original` and `scaled` values of `result` are set to `inValue` {63-64}. The `result` object is then written to `outChannel` {65}.

The remaining cases relate to the operation of the device in the normal state. If a `timer` alarm occurs the code associated with the `NORMAL_TIMER` case is obeyed {70}. The `timer`'s `timeout` alarm is reset for the next doubling period {71-72}. The scaling is multiplied by multiplier, which is two for doubling {73} as required by the device specification and an appropriate message printed {74}. The final case deals with inputs from `inChannel` {76}. The value is read from `inChannel` into `inValue` {77} and placed in the `original` property {79}of a new `result` object {78}. A scaled value is placed in the `scaled` property of a new `result` object {80}, which is then written to `outChannel` {81}.

5.1.3 The ScaledData Object

The `ScaledData` object is used to pass a pair of values from the `Scale` process to the `GPrint` process see Figure 5-1. Its structure is shown in Listing 6-3.

```
87     class ScaledData implements Serializable {

88       def int original
89       def int scaled

90       def String toString () {
91         def s = " " + original + "\t\t" + scaled
92         return s
93       }
94     }
```

Listing 5-4 The ScaledData Object

The properties of the object; `original` and `scaled` are defined {88, 89} and then a `toString`() method is defined {90-93} that is used when the object is printed.

More importantly, this is the first instance of user defined objects being communicated between processes. The first aspect to notice is there are no public data manipulation methods, other than implicit getters and setters that are created by the Groovy environment automatically, because in the parallel environment we encapsulate the data so that it is processed only within processes. It is not possible for one process to access another object's properties in another process to modify its state by calling public methods.

Concurrent processes pass object references over channels and thus a sending process has to guarantee that once it has written an object to a channel it does not modify that object in any way. This is most easily achieved by defining a new object instance for each write operation, see {62, 78}. In some cases, it may be necessary, for memory management reasons, to reuse an object and to ensure that a written object is not overwritten a deep copy is taken. An interface `JCSPCopy` which contains a single method `copy`() is provided in the `org.jcsp.groovy` package to facilitate this requirement. The programmer has to write the code to achieve the deep copy of the object. This can then be applied recursively to any nested objects.

If an object is to be passed between networked processes then a copy of the object is passed between the processes and so the object must implement the interface `serializable`. In this case it is not necessary to undertake the method `copy` because the serialization mechanism achieves this requirement.

5.1.4 Exercising the Scale Device Network

Listing 5-5 gives the script that implements the process network shown in Figure 5-1.

```
95     def data = Channel.createOne2One()
96     def timedData = Channel.createOne2One()
97     def scaledData = Channel.createOne2One()
98     def oldScale = Channel.createOne2One()
99     def newScale = Channel.createOne2One()
100    def pause = Channel.createOne2One()

101    def network = [ new GNumbers ( outChannel: data.out() ),
102                    new GFixedDelay ( delay: 1000,
103                                      inChannel: data.in(),
104                                      outChannel: timedData.out() ),
105                    new Scale ( inChannel: timedData.in(),
106                                outChannel: scaledData.out(),
107                                factor: oldScale.out(),
108                                suspend: pause.in(),
109                                injector: newScale.in(),
110                                scaling: 2 ),
111                    new Controller ( testInterval: 7000,
112                                     computeInterval: 700,
113                                     factor: oldScale.in(),
114                                     suspend: pause.out(),
115                                     injector: newScale.out() ),
116                    new GPrint ( inChannel: scaledData.in(),
117                                 heading: "Original      Scaled",
118                                 delay: 0)
119                  ]

120    new PAR ( network ).run()
```

Listing 5-5 Script to Exercise the Scale Device

All the output appears in the Eclipse console window with the messages from the `Scale` process intermingled with those from the output of the original and scaled data which appear in `GPrint`. The `delay` property {123} of `GPrint` is set to 0 so that any output is produced immediately. There is sufficient delay within the system caused by the `GFixedDelay` process to observe the process interactions.

5.2 Managing A Circular Queue Using Alternative Pre-conditions

A *queue* is a common data structure used in many applications. A number of cases have to be considered as follows.

1. data can only be put into the queue if there is space in the queue

2. data can only be taken from the queue if the queue is not empty

In a sequential implementation these states have to be tested before the queue can be manipulated and dealing with the situations where either a put or get to or from the queue cannot be undertaken can be problematic. A parallel implementation is much easier to design and specify because we can use an alternative with pre-conditions to ensure that operations only take place when it is safe. Figure 5-2 shows the basic structure that will be used to explain the operation of a queue.

The `QProducer` process puts a sequence of integers to the `Queue` process, where they are stored in a wrap-around List. The `QConsumer` process attempts to `get` data from the Queue, which if there is data available, is received by the process.

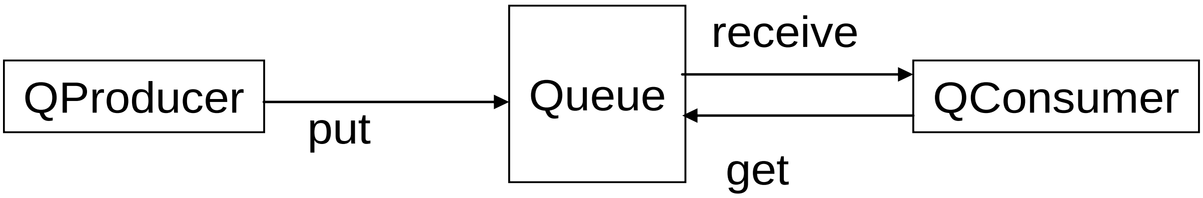

Figure 5-2 The Queue Process Network

5.2.1 QProducer and QConsumer Processes

The source of the `QProducer` process is given in Listing 5-6.

```
121    class QProducer implements CSProcess {

122      def ChannelOutput put
123      def int iterations = 100
124      def delay = 0

125      void run () {

126        def timer = new CSTimer()
127        println  "QProducer has started"

128        for ( i in 1 .. iterations ) {
129          put.write(i)
130          timer.sleep (delay)
131        }
132        put.write(null)
133      }
134    }
```

Listing 5-6 The QProducer Process script

The `timer` {126} is used to create a `delay` {130} between each `write` {129} to the `put` channel. A sequence of integers from 1 up to `iterations` {123} is output on the `put` channel. It should be noted that the `write` on the `put` channel may be delayed {129} if the queue has no available space. Once all the values have been written to the `put` channel a `null` value is also written (132) to indicate that processing has finished. This will be used to terminate the subsequent `Queue` and `QConsumer` processes.

The `QConsumer` process is specified in Listing 5-7. The use of the `timer` and associated `delay` {138, 140, 147} is the same as in `QProducer`. A Boolean `running` is defined {142} and is used to control the main loop of the process. The main loop of the process {143-151} initially writes a signal value of 1 on the `get` channel. The writing of this signal {144} may be delayed if the queue contains no available data. A value is `read` from the `receive` channel {145} into the object `v`. This `read` operation will take place immediately. The value that has been read is printed {146} after which the process may be delayed {147}. If the value read is `null` {148} then `running` is set to `false` {149} and the process will terminate at the next iteration of the `while` loop {143}.

```
class QConsumer implements CSProcess {

  def ChannelOutput get
  def ChannelInput receive
  def long delay = 0

  void run () {

    def timer = new CSTimer()
    println  "QConsumer has started"
    def running = true

    while (running) {
      get.write(1)
      def v = receive.read()
      println "QConsumer has read ${v}"
      timer.sleep (delay)
      if ( v == null ) {
         running = false
      }
    }
  }
}
```

Listing 5-7 The QConsumer Process

5.2.2 The Queue Process

The source for the `Queue` process is shown in Listing 5-8. The channel properties are defined {155-157} corresponding to Figure 5-2 and the size of the queue is specified in the property `elements` and is initially 5 {158}. The alternative associated with the `Queue` process is defined as `qAlt`, which has guards comprising the `put` and `get` channels {160}. A `Boolean` array, `preCon`, which has the same number of elements as there are guards in `qAlt`, is defined {161}. Two constants `PUT` and `GET` are defined {162, 163} that are used to index the `preCon` array and also to identify the cases in the `switch` statement associated with identifying the selected guard in the alternative.

The array `preCon` is used to record whether or not a new element can be put into the queue storage and similarly whether an element is available. Initially, therefore `preCon[PUT]` is set `true` {164} because there is bound to be space for a new element in the queue data structure because it must be empty. Similarly, `preCon[GET]` is set `false` {165} because there is no data available in the queue. The `List data` {166} provides the storage for the circular queue structure. The properties `count`, `front` and `rear` {167-169} record the state of the queue storage in terms of the number of data values in the queue, the location into which data can be added and removed from the queue respectively. The process is implemented as a loop {171-192}, which is controlled by a `Boolean running` {170} that is set `false` when a `null` value is communicated to the `QConsumer` process {183-185}.

The property `index` {172} indicates the alternative guard that has been selected. In order to be selected a guard must have its associated `preCon` element set to `true` and its channel must be enabled to read an input. Note how the pre-condition array is passed as a parameter to the alternative `priSelect` method {172}. A choice is then made depending upon which guard has been selected.

In the case of `PUT` the value `read` from `put` is placed in `data[front]` {175}. A message is then printed {176} and then the values of `count` and `front` are updated appropriately {177-178}. When `GET` is selected, the signal communication on the `get` channel is read and ignored {181}. The value in `data[rear]` is then written to channel `receive` {193}. The value in `data[rear]` is then tested to determine whether the `Queue` process should terminate {183-185}. After which, the values of `count` and `rear` are updated {186-187}. At the end of each loop of the queue process, the values stored in the elements of the `preCon` array are updated based upon the relative values of `count` and `elements` {190, 191}.

```
class Queue implements CSProcess {

  def ChannelInput  put
  def ChannelInput  get
  def ChannelOutput receive
  def int elements = 5

  void run() {

    def qAlt = new ALT ( [ put, get ] )
    def preCon = new boolean[2]
    def PUT = 0
    def GET = 1
    preCon[PUT] = true
    preCon[GET] = false
    def data = []
    def count = 0
    def front = 0
    def rear = 0

    def running = true
    while (running) {

      def index = qAlt.priSelect(preCon)
      switch (index) {

        case PUT:
          data[front] = put.read()
          println "Q: put ${data[front]} at ${front}"
          front = (front + 1) % elements
          count = count + 1
          Break

        case GET:
          get.read()
          receive.write( data[rear])
          if (data[rear] == null) {
               running = false
          }
          rear = (rear + 1) % elements
          count = count - 1
          break
      }

      preCon[PUT] = (count < elements)
      preCon[GET] = (count > 0 )
    }
    println "Q finished"
  }
}
```

Listing 5-8 The Queue Process Definition

The benefit of this alternative based formulation is that the pre-condition array modifies the behaviour of its underlying mechanism. Thus if the queue is full then `preCon[PUT]` is `false` and even if there is a communication on the `put` channel it will not be permitted. Similarly, if `preCon[GET]` is `false` then no signal on the `get` channel can be `read`, even if `QConsumer` has tried to write to it.

5.5 Summary

This chapter has explored the alternative mechanism together with its associated pre-condition Boolean array. It has shown by means of an example based upon a realistic system and one found in many program development applications that alternative has the ability to capture many aspects of real world systems and to provide a flexible means of modelling such systems.

5.6 Exercises

1. The accompanying web site contains a script, called TestQ, in package `ChapterExercises/src/c5` to run the queue network. The delays associated with `QProducer` and `QConsumer` can be modified. By varying the delay times demonstrate that the system works in the manner expected. Correct operation can be determined by the QConsumer process outputting the messages “`QConsumer has`

`read 1”` to `“QConsumer has read 50”` in sequence. What do you conclude from these experiments?

2. Reformulate the scaling device so that it uses pre-conditions rather than nested alternatives. Which is the more elegant formulation? Why?