4 Parallel Processes: Non Deterministic Input

In many systems there is a requirement to provide feedback from a down stream stage of the process to an upstream one. The upstream process has no idea when such a piece of feedback information is going to arrive and thus has to be able to accept it at any time. The behaviour of such a process is said to be non-deterministic because the arrival of the information cannot be determined when the process is defined. We know that such feedback can arrive but not when. Similarly, a process network may be subject to external interventions that change the parameters of the system. It is known that these interventions will occur but not when. This happens for example in medical laboratory equipment where a running check is kept of the range of values for each test that has been produced. Over a given period it is known that the mean value will lie within known bounds. If the machine is outwith those bounds then it enters an automatic recalibration process.

For this purpose *Alternative* provides the program structuring mechanism. In its simplest form the Alternative manages a number of input channels. On executing an Alternative the state of all the input channels is determined. If none of the channels are ready the Alternative waits until one is ready, reads the input and then obeys the code associated with that input. If one input is ready then that channel is read and its associated code is obeyed. If more than one channel is ready then one is chosen according to some selection criterion and the channel is read and its associated code obeyed. Typically, an Alternative is incorporated into a looping structure so that the input channels can be repeatedly accessed.

As a first example we shall take the process that generates a sequence of integers, `GNumbers`, previously described in Section 3.4. We shall modify it so that it accepts an input which resets the sequence to a number input by the user at any time chosen by the user.

4.1 Reset Numbers

The structure of the revised numbers process is shown in Figure 4-1.

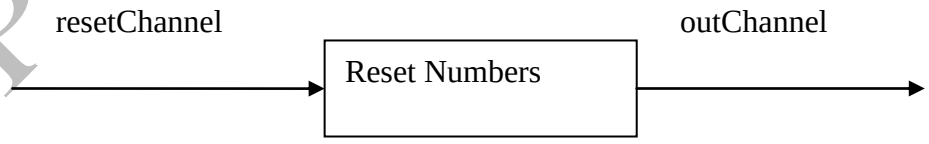

Figure 4-1 The Reset Numbers Process Structure

This however does not indicate the changes required to the internal operation of the ResetNumbers process, which is shown in Figure 4-2.

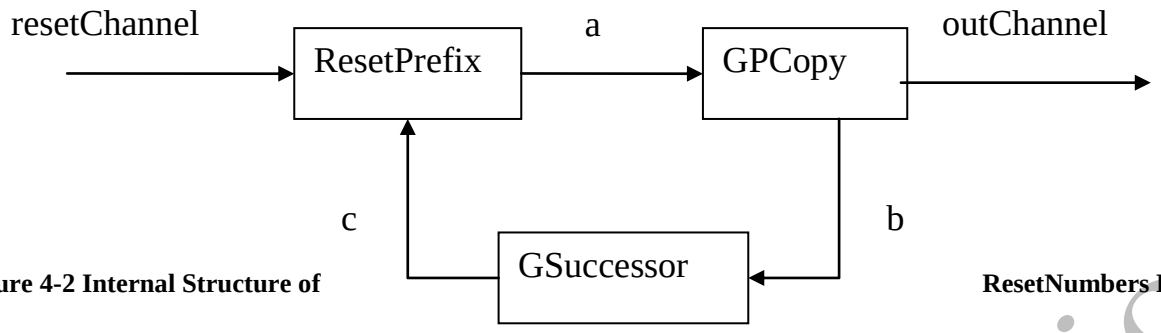

Figure 4-2 Internal Structure of ResetNumbers Process

Instead of using the `GPrefix` process of `GNumbers` (see 3.4) we use a new process `ResetPrefix`. Figure 4-2 exposes the fact that the `ResetPrefix` process contains two input channels over which it can alternate. The coding of the `ResetPrefix` process is shown in Listing 4-1.

```
01    class ResetPrefix implements CSProcess {

02      def int prefixValue = 0
03      def ChannelOutput outChannel
04      def ChannelInput  inChannel
05      def ChannelInput  resetChannel

06      void run () {
07        def guards = [ resetChannel, inChannel  ]
08        def alt = new ALT ( guards )
09        outChannel.write(prefixValue)

10        while (true) {
11          def index = alt.priSelect()

12          if (index == 0 ) {    // resetChannel input
13            def resetValue = resetChannel.read()
14            def inputValue = inChannel.read()
15            outChannel.write(resetValue)
16          }

17          else {    //inChannel input
18            def inputValue = inChannel.read()
19            outChannel.write(inputValue)
20          }
21        }
22      }
23    }
```

Listing 4-1 ResetPrefix Coding

The properties of the process comprise {2-5}; the initial `prefixValue` from which the first sequence will start. The channels it uses to communicate, namely, `inChannel`, `outChannel` and `resetChannel`. The latter receives the value to which the sequence of integers is to be reset.

An `Alternative` comprises a number of *guards* and associated *guarded commands*. In this case a guard is simply an input channel and the guarded command is the code that is associated with that channel input. There are two `guards`{7}, formed as a `List`; the `resetChannel` and `inChannel`. The latter is used during the normal operation of the process network. The order in which these guards are specified is important because we wish to give priority to the `resetChannel`. The alternative `alt` is defined {8} by means of the `ALT` helper class, which takes a `List` of guards (see Appendix 2).

The fundamental operation of the process remains the same as `GPrefix`, in that the value of `prefixValue` is output {9} after which the process repeatedly {10} inputs a value and then outputs the same value on its `outChannel`. In this case the input comes from either the `resetChannel` or the `inChannel`. The method `priSelect()` applied to `alt` {11} returns the `index` of the channel that was selected from the

alternative using the criteria that the channel selected has the lowest `List` index if more than one guard is ready. Thus an `index` value of 0 implies that the `resetChannel` is ready to be read from and 1 implies that `resetChannel` was not ready and `inChannel` was ready. Hence we can construct a simple `if` statement {12-20} to discriminate these cases for each of the guarded commands. In more complex alternatives, with more guards, we would use a `switch` statement. The first operation of any guarded command sequence must be to read from the channel that was selected by the alternative.

The guarded command for the `resetChannel` reads from the channel {13} and assigns its value to `resetValue`. The value currently circulating round the network needs to be read from `inChannel` {14} and ignored after which `resetValue` can be written to the `outChannel` {15}. The guarded command for input from `inChannel` is identical to the original version of `GPrefix` in that a value is read from `inChannel` into `inputValue` {18} and then written to the `outChannel` {18}.

Listing 4-2 shows the coding of the `ResetNumbers` process, which can be seen to be a direct implementation of Figure 4-2.

```
24    class ResetNumbers implements CSProcess {

25      def ChannelOutput outChannel
26      def ChannelInput resetChannel
27      def int initialValue = 0

28      void run() {

29        One2OneChannel a = Channel.createOne2One()
30        One2OneChannel b = Channel.createOne2One()
31        One2OneChannel c = Channel.createOne2One()

32        def testList = [ new ResetPrefix ( prefixValue: initialValue,
33                                           outChannel: a.out(),
34                                           inChannel: c.in(),
35                                           resetChannel: resetChannel ),

36                         new GPCopy ( inChannel: a.in(),
37                                      outChannel0: outChannel,
38                                      outChannel1: b.out() ),

39                         new GSuccessor ( inChannel: b.in(),
40                                          outChannel: c.out())
41                       ]

42        new PAR ( testList ).run()
43      }
44    }
```

Listing 4-2 Definition of The ResetNumbers Process

4.2 Exercising ResetNumbers

In order to exercise `ResetNumbers` a process is required that can send values to its `resetChannel`. This is most simply achieved by running two processes in parallel, each with their own user interface so the interaction between the processes can be observed. The structure of this process network is shown in Figure 4-3. The user interface is implemented using a GConsole process, see Appendix 4.

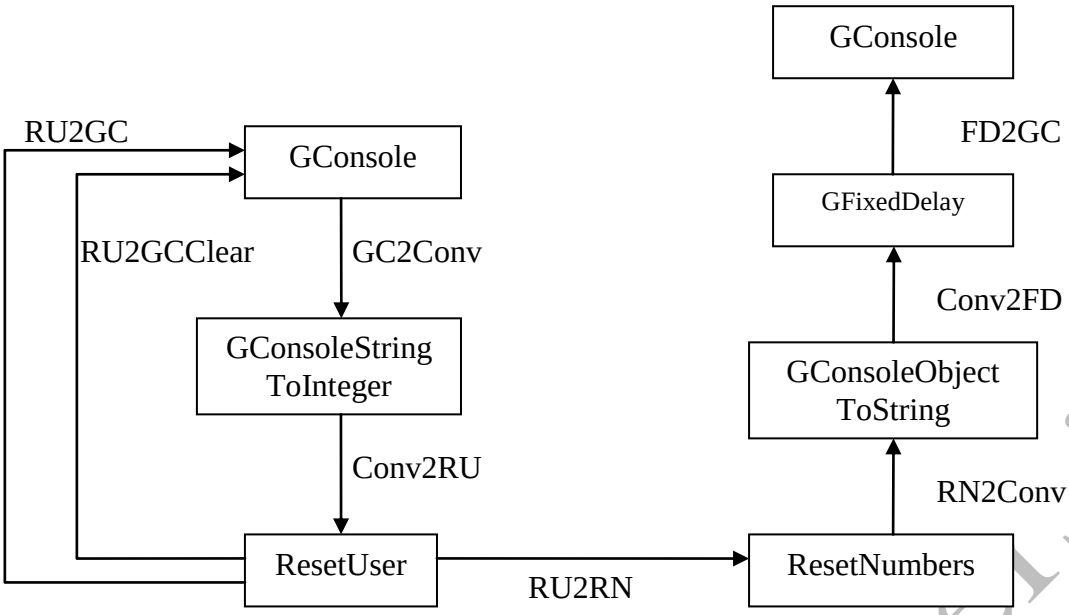

Figure 4-3

Exercising the ResetNumbers Process

The `ResetUser` process receives inputs from its `GConsole` user interface process through a `GConsoleStringToInteger` process, which coverts an input string typed into the input area of the user interface into an `Integer`. The channel `RU2GC` is used to output messages to the user interface's output area. The channel `RU2GCClear` is used to clear the user interface's input area between inputs. On receiving an input, the `ResetUser` process outputs this value to the `ResetNumbers` process using the channel `RU2RN`. This value is input on the `ResetNumber`'s `resetChannel` (see Figure 4-2 and Listing 4-2), which is then communicated to the `resetChannel` of the `resetPrefix` process (see Listing 4-1 {5}) contained within the `ResetNumbers` process.

The process `ResetNumbers` outputs a sequence of numbers, the content of which changes as reset values are read, to its `RN2Conv` channel. The process `GConsoleObjectToString` process converts any `object` to a `String` using the object's `toString()` method. The `String` representation of the values passes through a process called `GFixedDelay` which introduces a delay into the output stream sent to the `GConsole` process so that it is easier to read from the user interface's output area.

Listing 4-3 presents the coding of the `ResetUser` process. An initial message is written to the user interface {51} after which values are repeatedly {52} read from the `GConsoleStringToInteger` process as an `Integer` value `v` {53}, after which the input area of the user interface is cleared {54}. The value to be sent to the reset channel is then written to the channel `resetValue` {55}.

```
45     class ResetUser implements CSProcess {

46       def ChannelOutput resetValue
47       def ChannelOutput toConsole
48       def ChannelInput fromConverter
49       def ChannelOutput toClearOutput

50       void run() {
51            toConsole.write( "Please input reset values\n" )

52            while (true) {
53              def v = fromConverter.read()
54              toClearOutput.write("\n")
55              resetValue.write(v)
56            }
57       }
58     }
```

Listing 4-3 The ResetUser Process

Listing 4-4 shows the implementation of the process network shown in Figure 4-3.

```
59     One2OneChannel RU2RN = Channel.createOne2One()

60     One2OneChannel RN2Conv = Channel.createOne2One()
61     One2OneChannel Conv2FD = Channel.createOne2One()
62     One2OneChannel FD2GC = Channel.createOne2One()

63     def RNprocList = [ new ResetNumbers ( resetChannel: RU2RN.in(),
64                                           initialValue: 1000,
65                                           outChannel: RN2Conv.out() ),
66                        new GObjectToConsoleString  ( inChannel: RN2Conv.in(),
67                                                      outChannel:Conv2FD.out() ),
68                        new GFixedDelay ( delay: 200,
69                                          inChannel: Conv2FD.in(),
70                                          outChannel: FD2GC.out() ),
71                        new GConsole ( toConsole: FD2GC.in(),
72                                       frameLabel: "Reset Numbers Console"  )
73                      ]

74     def One2OneChannel RU2GC = Channel.createOne2One()
75     def One2OneChannel GC2Conv = Channel.createOne2One()
76     def One2OneChannel Conv2RU = Channel.createOne2One()
77     def One2OneChannel  RU2GCClear= Channel.createOne2One()

78     def RUprocList = [ new ResetUser ( resetValue: RU2RN.out(),
79                                        toConsole: RU2GC.out(),
80                                        fromConverter: Conv2RU.in(),
81                                        toClearOutput: RU2GCClear.out()),
82                        new GConsoleStringToInteger ( inChannel: GC2Conv.in(),
83                                                      outChannel: Conv2RU.out()),
84                        new GConsole ( toConsole: RU2GC.in(),
85                                       fromConsole: GC2Conv.out(),
86                                       clearInputArea: RU2GCClear.in(),
87                                       frameLabel: "Reset Value Generator" )
88                      ]

89     new PAR (RNprocList + RUprocList).run()
```

Listing 4-4 The Script That Exercises the RestNumbers Process

Initially, the channel `RU2RN` is defined {59}. After which the two parts of the network are defined separately. First, the `ResetNumbers` network is defined {60-73}. The channels `RN2Conv`, `Conv2FD` and `FD2GC` are defined {60-62}. The list `RNprocList` contains instances of each of the processes shown in Figure 4-3 used to implement the `ResetNumbers` process and its interface components {63}. By inspection, it can be seen the connections shown in Figure 4-3 are implemented by the properties passed to each of the processes in the list {63-73}. The processes `GObjectToConsoleString`, `GFixedDelay` and `GConsole` are described in more detail in Appendix 4. The timing of the stream of output numbers is governed by the `delay` property of the `GFixedDelay` process {68}.

The second part of Lisiting 4-4 {74-88} shows the network used to implement the `RUprocList` that implement the `ResetUser` part of the system and its interface components. The channels RU2GC, GC2Conv, Conv2RU and RU2GCClear implement the channels shown in Figure 4-3 {74-77}. Finally, the two process lists RNprocList and RUprocList are concatenated using the overloaded + operator {89} and the network is executed. Each of the parts of the network has their own `GConsole` process. The `frameLabel` property of this process is used to write a title on each of the user interface windows {72, 87} respectively.

When the processes are invoked it can be observed that as reset values are typed into the `Reset Value Generator` console, the values in the `Reset Numbers Console` continue for a short time with the original sequence and then produce a sequence starting with the recently typed reset value.

4.3 Summary

This chapter has introduced the concept of the alternative and shown how it can be used to choose between a number of input channels. In the next chapter we shall show how the guards can be extended to include timer alarms in a more realistic example derived from machine tool control.

4.4 Exercises

1. What happens if line {14} of `ResetPrefix` is commented out? Why?

1. Construct a different formulation of `ResetNumbers` that connects the reset channel to the `GSuccessor` process instead of `GPrefix`. You will have to write a `ResetSuccssor` process. Does it overcome the problem identified in Exercise 1? If not why not?