3 Process Networks: Build It Like Lego[1]

One of the main advantages of the CSP based approach we are using is that processes can be combined using a simple compositional style. It is very much *what you see is what you get*!

In arithmetic the meaning of the composition 1 + 2 + 3 is immediately obvious and results in the answer 6. The composition of processes is equally simple and obvious. Thus we can build a set of basic building block processes[2], like Lego bricks, from which we can construct larger systems, the meaning of which will be obvious given our understanding of the basic processes.

All of the building block processes are to be found in the package `org.jcsp.groovy.plugAndPlay`. A more detailed discussion of these processes is to be found in Appendix 4.

3.1 Prefix Process

The process diagram of `GPrefix` is given in Figure 3-1 and its definition is presented in Listing 3-1. `GPrefix` initially outputs the `prefixValue` on its `outChannel` {7} and thereafter it writes everything it reads on its `inChannel` {9} to its `outChannel`, using a non-terminating loop {8-10}.

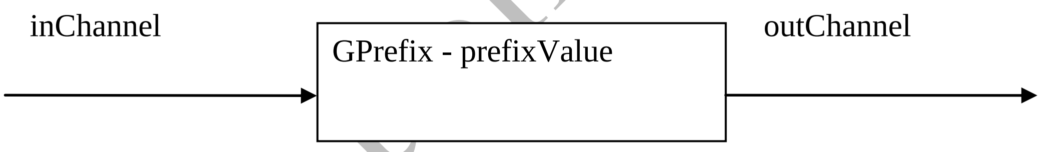

Figure 3-1 GPrefix Process Diagram

The `GPrefix` process has an input channel `inChannel` and an output channel `outChannel`, which are properties of the process {4, 5}. In addition, there is a property called `prefixValue` that has the initial value 0 {3}, which can be changed when a process instance is created.

---

[1] Lego is a registered trademark of the LEGO Group, www.lego.com

[2] The basic processes are based upon those in the package org.jcsp.plugNplay. This is denoted by the G in process name

```
01     import org.jcsp.lang.*

02     class GPrefix implements CSProcess {

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

06       void run () {
07         outChannel.write(prefixValue)
08         while (true) {
09           outChannel.write( inChannel.read() )
10         }
11       }
12     }
```

Listing 3-1 GPrefix Process Definition

3.2 Successor Process

The process diagram for `GSuccessor` is shown in Figure 3-2 and its coding in Listing 3-2. The process simply {19} reads in a value on its `inChannel` and then writes this value plus 1 to its `outChannel`. It does this in a `while` loop that never terminates {18-20}.

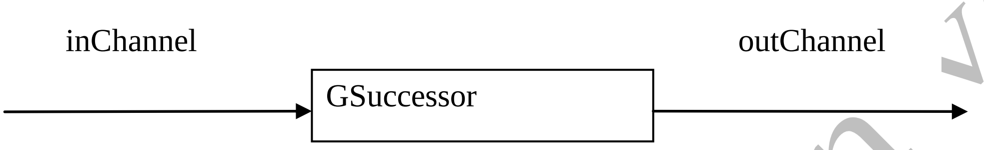

Figure 3-2 GSuccessor Process Diagram

```
13     import org.jcsp.lang.*

14     class GSuccessor implements CSProcess {

15       def ChannelInput inChannel
16       def ChannelOutput outChannel

17       void run () {
18         while (true) {
19           outChannel.write( inChannel.read() + 1 )
20         }
21       }
22     }
```

Listing 3-2 GSuccessor Process Definition

3.3 Parallel Copy

The process diagram for `GPCopy` is given in Figure 3-3 and its coding in Listing 3-3. The process inputs a value on its `inChannel` {27}, which it outputs to `outChannel0` {28} and `outChannel1` {29} in parallel. This is repeated forever. By outputting to its output channels in parallel we are assured that it does not matter the order in which these channels are read by the corresponding input channels. We are also guaranteed that a read on its input channel will not take place until both the outputs have completed because a parallel (PAR) does not terminate until all its constituent processes have terminated.

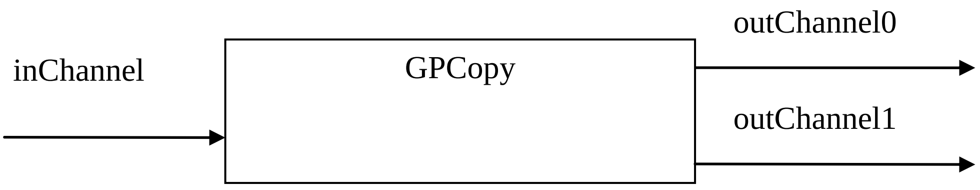

Figure 3-3 Process Diagram of GPCopy

```
23      import org.jcsp.plugNplay.ProcessWrite
24      import org.jcsp.lang.*
25      import org.jcsp.groovy.*

26      class GPCopy implements CSProcess {

27        def ChannelInput inChannel
28        def ChannelOutput outChannel0
29        def ChannelOutput outChannel1

30        void run () {

31          def write0 = new ProcessWrite ( outChannel0)
32          def write1 = new ProcessWrite ( outChannel1)
33          def parWrite2 = new PAR ( [ write0, write1 ] )

34          while (true) {
35            def i = inChannel.read()
36            write0.value = i
37            write1.value = i
38            parWrite2.run()
39          }
40        }
41      }
```

Listing 3-3 GPCopy Process Definition

`GPCopy` utilises the process `ProcessWrite` from `org.jcsp.plugNplay`, demonstrating that we can incorporate previously written Java processes into the Groovy environment. Two instances of `ProcessWrite` are defined {31, 32} each accessing one of the output channels. A `PAR` of the two processes is then defined {33} called `parWrite2`, which is not run at this time. An instance of `ProcessWrite` has a publicly available field called `value` that is assigned the data to be written.

The non-terminating loop {34-39} firstly reads in a value from the `inChannel` {35}, the value of which is assigned to the `value` fields of the two `ProcessWrite` instances, `write0` {36}and `write1` {37}. The parallel `parWrite2` is then run {38}, which causes the writing of the value read in from `inChannel` to `outChannel0` and `outChannel1` in parallel, after which it terminates. `ProcessWrite` terminates as soon as it has written a single value to its output channel. Once `parWrite2` has terminated, processing resumes at the `while` loop {34}.

3.4 Generating a Sequence of Integers

The three processes, `GPrefix`, `GSuccessor` and `GPCopy` can be combined to form a network that outputs a sequence of integers on `outChannel` as shown in Figure 3-4 and Listing 3-4.

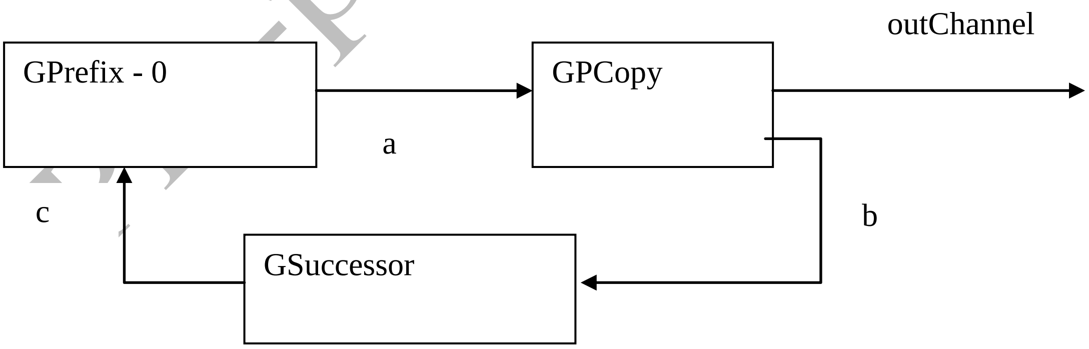

Figure 3-4 Process Network Diagram to Generate a Stream of Integers (GNumbers)

```
42     import org.jcsp.lang.*
43     import org.jcsp.groovy.*

44     class GNumbers implements CSProcess {

45       def ChannelOutput outChannel

46       void run() {

47         One2OneChannel a = Channel.createOne2One()
48         One2OneChannel b = Channel.createOne2One()
49         One2OneChannel c = Channel.createOne2One()

50         def numbersList =  [ new GPrefix    ( prefixValue: 0,
51                                               inChannel: c.in(),
52                                               outChannel: a.out() ),

53                              new GPCopy     ( inChannel: a.in(),
54                                               outChannel0: outChannel,
55                                               outChannel1: b.out() ),

56                              new GSuccessor ( inChannel: b.in(),
57                                               outChannel: c.out())
58                            ]

59         new PAR ( numbersList ).run()
60       }
61     }
```

Listing 3-4 Definition of the GNumbers Process

The `GNumbers` process has a single output channel `outChannel` property {45} upon which the stream of integers is output. Three internal channels `a`, `b` and `c` are defined {47-49} as `One2OneChannel` interfaces and these are used to connect the processes together in a manner that directly reflects the process network diagram, Figure 3-4. Note that in Groovy we do not need to specify the types of the channels but for explanation and additional type checking this is done to aid clarity. For example, the two output channels of `GPCopy` are assigned to the property `outChannel` and `b.out()` while its input channel is assigned to `a.in()`.

The design process becomes one of creating a process network diagram and then using that to define the required channels which are then used to connect the processes together. The system is able to check, using the interface specifications, that an input end of a channel specified by the `in()` method is connected to a `ChannelInput` and similarly for output channels because we have specified the types of the channels in the properties of the process class definitions.

3.5 Testing GNumbers

Figure 3-5 shows the process network that can be used to test the operation of the process `GNumbers`. It is apparent that the easiest way of testing the process `GNumbers` is to print the stream of numbers to the console window. For this purpose a `GPrint` process is provided. `GPrint` has a `ChannelInput` for reading the stream of numbers as the property `inChannel`. It also has a property, `heading`, that is a `String`, which contains a title for the printed stream. The corresponding script for the network shown in Figure 3-5 is given in Listing 3-5.

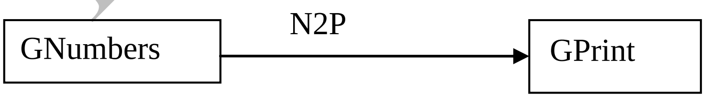

Figure 3-5 Network to Test GNumbers

```
62     import org.jcsp.lang.*
63     import org.jcsp.groovy.*

64     One2OneChannel N2P = Channel.createOne2One()

65     def testList = [ new GNumbers ( outChannel: N2P.out() ),

66                      new GPrint   ( inChannel: N2P.in(),
67                                     heading : "Numbers" )
68                    ]

69     new PAR ( testList ).run()
```

Listing 3-5 The Script to Test GNumbers

A single channel is created {64} called `N2P` that is used to connect `GNumbers` to `GPrint`. The list of processes is created {65-68} with the properties assigned to the input and output ends of `N2P` and the `heading` property of `GPrint` is set to "Numbers". A typical output is shown in Output 3-1. It is noted that the user has to terminate the system by interrupting the console stream using the Eclipse terminate program button. The processes have been constructed using never ending while-loops and thus run forever, unless otherwise terminated.

```
Numbers
0
1
2
3
4
5
6
7
8
9
10
11
```

Output 3-1 Output from the Script Test GNumbers

3.6 Creating a Running Sum

We will now use the output from `GNumbers` as input to a process called `GIntegrate` that reads a stream of integers and outputs the running sum of the numbers read so far, as another stream of numbers. In order to do this we shall need a process that undertakes addition of numbers arriving in a stream of such numbers. The `GPlus` process does this and its coding is shown in Listing 3-6.

```
70     import org.jcsp.plugNplay.ProcessRead
71     import org.jcsp.lang.*
72     import org.jcsp.groovy.*

73     class GPlus implements CSProcess {

74       def ChannelOutput outChannel
75       def ChannelInput inChannel0
76       def ChannelInput inChannel1

77       void run () {

78         ProcessRead read0 = new ProcessRead ( inChannel0)
79         ProcessRead read1 = new ProcessRead ( inChannel1)
80         def parRead2 = new PAR ( [ read0, read1 ] )

81         while (true) {
82           parRead2.run()
83           outChannel.write(read0.value + read1.value)
84         }
85       }
86     }
```

Listing 3-6 GPlus process coding

The `GPlus` process uses techniques similar to that used in `GPCopy`, except that we read from two input channels in parallel using the process `ProcessRead`, which reads a single value from a channel and then terminates. `GPlus` has two input channels, `inChannel0` and `inChannel1` {75, 76} and one output channel, `outChannel` {74} upon which the sum of the two inputs are written. Two `ProcessRead` processes are constructed called `read0` {78} and `read1` {79} and these are used to construct a `PAR` called `parRead2` {80}. The main loop of the process {81-84} initially invokes the parallel `parRead2` {82}. This parallel only terminates when both `read0` and `read1` have read a value and terminated. The values read are obtained from a publicly available field, `value`, of a `ProcessRead`. The two values are added together and then written to the output channel {83}.

Listing 3-7 gives the coding for the process `GIntegrate` and its associated process network diagram is given in Figure 3-6. The coding can be seen to be a representation of the diagram in the same way as previous transformations of diagrams into codings.

The operation of `GIntegrate` proceeds as follows. The process `GPrefix` can output its initial value, 0, which forms one of the inputs to `GPlus`, using channel `c`. The other input from `GPlus` is read from `GIntegrate`'s `inChannel`. The `GPlus` process and hence the `GIntegrate` process will now wait until there is an input on the `inChannel`. Once this arrives the addition of the two values will take place and the result written to the channel `a`, which forms the input to `GPCopy`. `GCopy` can now output the current sum on the `outChannel` and also send a copy to `GPrefix`, using channel `b`, which immediately outputs the value unaltered to the channel `c`. In this way the current running sum is circulated around the network and is also output to a subsequent process.

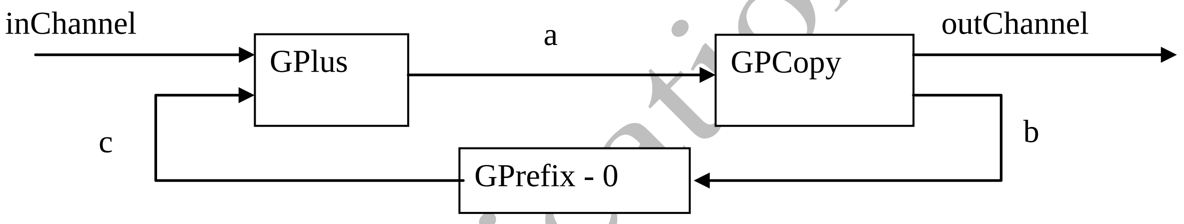

Figure 3-6 Process Network Diagram of GIntegrate

```
import org.jcsp.lang.*
import org.jcsp.groovy.*

class GIntegrate implements CSProcess {

  def ChannelOutput outChannel
  def ChannelInput  inChannel

  void run() {

    One2OneChannel a = Channel.createOne2One()
    One2OneChannel b = Channel.createOne2One()
    One2OneChannel c = Channel.createOne2One()

    def integrateList = [ new GPrefix ( prefixValue: 0,
                                        outChannel: c.out(),
                                        inChannel: b.in() ),

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

                          new GPlus   ( inChannel0: inChannel,
                                        inChannel1: c.in(),
                                        outChannel: a.out() )
                        ]

    new PAR ( integrateList ).run()
  }
}
```

Listing 3-7 GIntegrate Process Definition

A process network to test the operation of `GIntegrate`, by outputting the current value of running sum is presented in Figure 3-7. `GNumbers` provides the input stream into `GIntegrate` using the channel `N2I` and the output from `GIntegrate` is written, using the channel `I2P`, to the `GPrint` process which writes the stream of numbers to the console.

The script that invokes this network is shown in Listing3-8. The script is taken directly from the process network diagram by connecting the output and input ends of each of the channels, `N2I` and `I2P`, to the appropriate property of the processes.

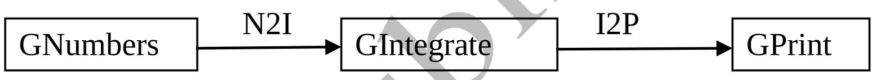

Figure 3-7 The Process Network to Demonstrate the Operation of GIntegrate

```
import org.jcsp.lang.*
import org.jcsp.groovy.*

One2OneChannel N2I = Channel.createOne2One()
One2OneChannel I2P = Channel.createOne2One()

def testList = [ new GNumbers   ( outChannel: N2I.out() ),

                 new GIntegrate ( inChannel: N2I.in(),
                                  outChannel: I2P.out() ),

                 new GPrint     ( inChannel: I2P.in(),
                                  heading: "Integrate")
               ]

new PAR ( testList ).run()
```

Listing 3-8 Script that Implements the Network of Figure 3-7

Output 3-2 shows the console window after the network has been allowed to execute for a short period of time. It can be seen by observation that each output is the sum of the numbers so far from the sequence 0, 1, 2, … . Later, we shall see how we can output all the intermediate values.

```
Integrate
0
1
3
6
10
15
21
28
36
45
```

Output 3-2 Running Sum Generated by the Sequence of Positive Integers

3.7 Generating the Fibonacci Sequence

The Fibonacci sequence comprises; 0, 1, 1, 2, 3, 5, 8, 13, 21, …$f_{n-2}+f_{n-1}$, …. The first two numbers in the sequence f_0 and f_1 have to be predefined and are typically set to 0 and 1 but could be any value. It can be seen that we need to create the first two numbers in the sequence and we already have a process, `GPrefix` that achieves this. We now need a process that will read two numbers, in sequence and then output the sum of the pair of numbers. The next iteration will take the second number in the sequence and pair it to the third number that is input, output their sum and so on.

3.7.1 Adding Pairs of Numbers

Listing 3-9 gives the definition of a process that inputs a stream of numbers and outputs another stream which contains the sum of pairs of numbers. The process `GStatePairs` initially reads in two numbers from the input stream, `inChannel`, {156, 157} then, within a loop outputs their sum {159} to `outChannel`, copies the second number to the first {160}and then reads another number `n2` from `inChannel` {161}.

```
120    class GStatePairs implements CSProcess {

121      def ChannelOutput outChannel
122      def ChannelInput  inChannel

123      void run() {

124        def n1 = inChannel.read()
125        def n2 = inChannel.read()

126        while (true) {
127          outChannel.write ( n1 + n2 )
128          n1 = n2
129          n2 = inChannel.read()
130        }
131      }
132    }
```

Listing 3-9 Process GStatePairs

The process network diagram that implements the generation of the Fibonacci sequence is shown in Figure 3-8 and its associated process definition is shown in Listing 3-10.

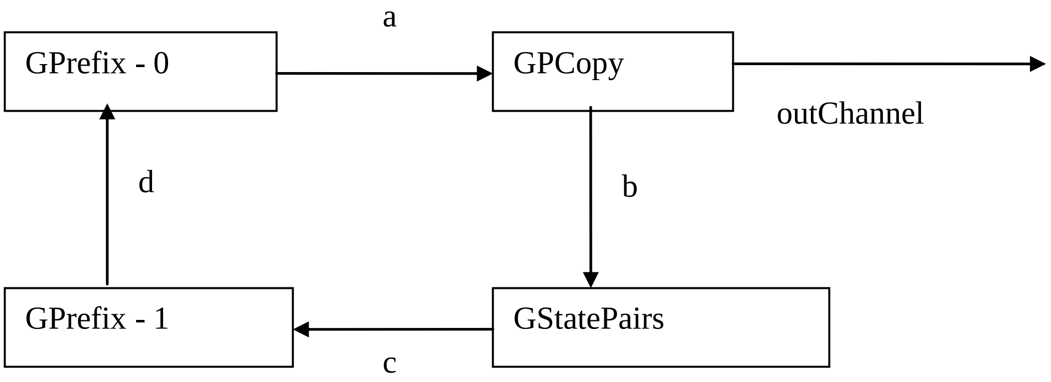

Figure 3-8 Process Network Diagram to Generate the Fibonacci Sequence

Initially, `GPrefix-0` is the only process that can run because it is the only one that can undertake an output. `GPCopy` is waiting for an input as is `GStatePairs`. `GPrefix-1` is trying to output and will not be able to, until `GPrefix-0` reads from its input channel, which it will do once it has written the 0 to `GPCopy`.

It can be seen, by inspection, that the code given in Listing 3-10, directly implements the process network diagram given in Figure 3-8. The four channels, `a`, `b`, `c` and `d` are defined {136-139}. The list of processes is then created as `testList` {140-151} comprising four elements, one for each of the required processes. The list of processes is then invoked using a `PAR` {152}.

```
133    class FibonacciV1 implements CSProcess {

134      def ChannelOutput outChannel

135      void run () {

136        One2OneChannel a  = Channel.createOne2One()
137        One2OneChannel b  = Channel.createOne2One()
138        One2OneChannel c  = Channel.createOne2One()
139        One2OneChannel d  = Channel.createOne2One()

140        def testList = [ new GPrefix ( prefixValue: 0,
141                                       inChannel: d.in(),
142                                       outChannel: a.out() ),

143                         new GPrefix ( prefixValue: 1,
144                                       inChannel: c.in(),
145                                       outChannel: d.out() ),

146                         new GPCopy  ( inChannel: a.in(),
147                                       outChannel0: b.out(),
148                                       outChannel1: outChannel ),

149                         new GStatePairs ( inChannel: b.in(),
150                                           outChannel: c.out() ),
151                       ]

152        new PAR ( testList ).run()
153      }
154    }
```

Listing 3-10 Fibonacci Process Definition

Listing 3-11 shows the script by which the output from the Fibonacci system can be produced using the previously defined `GPrint` process.

```
One2OneChannel F2P = Channel.createOne2One()

def testList = [ new FibonacciV1 ( outChannel: F2P.out() ),

                 new GPrint      ( inChannel: F2P.in(),
                                   heading: "Fibonacci V1")
               ]

new PAR ( testList ).run()
```

Listing 3-11 The Script to Output the Fibonacci Sequence

The output from this script is shown in Output 3-3.

```
Fibonacci V1
0
1
1
2
3
5
8
13
21
34
55
89
```

Output 3-3 Console Output from Script Generating the Fibonacci Sequence

There is, however, a problem with this solution because we now have a process definition for `GStatePairs` (Listing 3-9) that contains some state (`n1` and `n2`) that is retained between iterations of the process. All the other process defined so far, contain no such state. We have also defined a process `GStatePairs` that does addition within it and yet we have already defined a process `GPlus` (Listing 3-6) that undertakes stateless addition. How can we build another process that enables us to use the `GPlus` process and which yet can be used to create the affect of `GStatePairs`? This may seem a somewhat esoteric argument but processes that contain state are much more difficult to modify should changes be required in future, especially if it is desired to modfy their behaviour dynamically. This is discussed in the next chapter.

3.7.2 Using `GPlus` to Create the Sum of Pairs of Numbers

In order to use `GPlus` we need two input streams comprising the numbers to be added together. We can use `GPCopy` to copy the input stream, which would give us two identical streams. We however require adding the current number to the previous one. Hence we require a process that removes the first number from one of the streams and then just outputs what it inputs. If this process is inserted into one of the streams coming from `GPCopy` then we will create that stream with the current number and the other will, in fact contain the previous number. This is shown in Figure 3-9, where the process `GTail` is introduced.

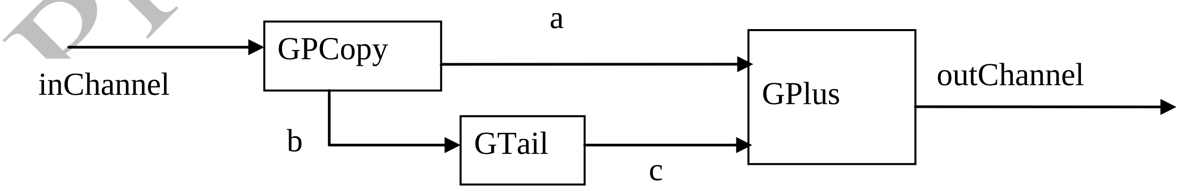

Figure 3-9 GPairs Process Network that Adds Pairs of Numbers using GPlus

The definition of `GTail` is shown in Listing 3-12. The first value sent to `inChannel` is `read` but not retained {165}. Thereafter, objects are `read` from `inChannel` and immediately `written` to `outChannel` {167}. This formulation retains no state between iterations of the loop {166-168}.

```
class GTail implements CSProcess {

  def ChannelOutput outChannel
  def ChannelInput inChannel

  void run () {
    inChannel.read()
    while (true) {
      outChannel.write( inChannel.read() )
    }
  }
}
```

Listing 3-12 Definition of GTail

The operation of the network given in Figure 3-9 is as follows; the first number, 0, is read by `GPCopy` and copied to channels `a` and `b` in parallel. `GTail` reads the 0 on channel `b` and ignores it! Meanwhile the output on channel `a` is read by `GPlus`. `GPCopy` now reads the next number, 1, and attempts to copy this to both channels `a` and `b` in parallel. That to channel `b` will be read by `GTail` and immediately output to channel `c` to be read by `GPlus`, which can now do the addition and subsequent output of the sum of 0 and 1. `GPCopy` is now able to write the copy of 1 to the channel `a` as `GPlus` is now ready to read, in parallel. The system continues in this manner, with none of the processes retaining any state and simply relying on the fact that processes input from and output to multiple channels in parallel and that the order in which the communications takes places does not matter. The semantics of channel communication ensure that no data is lost.

The coding of the stateless version of the process, `GPairs`, to add pairs of numbers from a stream is shown in Listing 3-13 and follows the structure shown in Figure 3-9.

```
class GPairs implements CSProcess {

  def ChannelOutput outChannel
  def ChannelInput  inChannel

  void run() {

    One2OneChannel a = Channel.createOne2One()
    One2OneChannel b = Channel.createOne2One()
    One2OneChannel c = Channel.createOne2One()

    def pairsList  = [ new GPlus   ( outChannel: outChannel,
                                     inChannel0: a.in(),
                                     inChannel1: c.in() ),

                       new GPCopy  ( inChannel: inChannel,
                                     outChannel0: a.out(),
                                     outChannel1: b.out() ),

                       new GTail   ( inChannel: b.in(),
                                     outChannel: c.out() )
                       ]

    new PAR ( pairsList ).run()
  }
}
```

Listing 3-13 The GPairs Process Definition

The definition of the second version of the Fibonnaci process is the same as that given in Listing 3-10 with line 149-150 replaced with the invocation of the constructor for `GPairs` instead of `GStatePairs`. The execution of the second version is the same as that shown in Listing 3-11 with line 156 creating and instance of the second version of the Fibonnaci process rather than the first. The output is identical from both systems.

3.7.3 Lessons Learned

We should always try to reuse existing processes whenever possible and that often the best way of solving a problem is to define another process rather than changing or extending an existing one. In other words, if we try to keep each process as simple as possible and to compose systems from lots of small, easily understood processes it will be easier to argue about the behaviour of the complete network.

3.8 Generating Squares of Numbers

In this example, we will reuse the processes we have created so far to create a sequence of squares of numbers. The process network to achieve this is shown in Figure 3-10 and the corresponding Listing 3-14 gives the process definition. The process simply writes to its `outChannel` the squares of the numbers starting with 1 upwards. It can be tested by connecting the `outChannel` to a `GPrint` process.

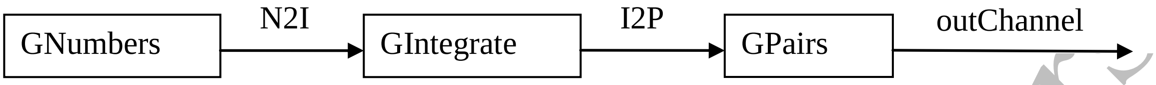

Figure 3-10 The GSquares Process Network

By inspection it can be seen that the `GSquares` process Listing 3-14 does implement the network given in Figure 3-10. However, what is not obvious is how this result is achieved. To try to understand this we need to print the output from each stage of the squares process. For this we require a process that prints a number of parallel inputs.

```
190    class GSquares implements CSProcess {

191      def ChannelOutput outChannel

192      void run () {

193        One2OneChannel N2I = Channel.createOne2One()
194        One2OneChannel I2P = Channel.createOne2One()

195        def testList =  [ new GNumbers   ( outChannel: N2I.out() ),

196                          new GIntegrate ( inChannel: N2I.in(),
197                                             outChannel: I2P.out() ),

198                          new GPairs     ( inChannel: I2P.in(),
199                                             outChannel: outChannel ),
200                        ]

201        new PAR ( testList ).run()
202      }
203    }
```

Listing 3-14 GSquares Process Definition

3.9 Printing in Parallel

There are many occasions in which we wish to print output from a set of parallel processes so that the output correlates the state of each process at a consistent point in their execution. The `GParPrint` process achieves this by reading a number of inputs in parallel and then printing out each in a tabular manner one set of inputs to a line of text. Its coding is shown in Listing 3-15.

The property `inChannels` {208} is of type `ChannelInputList`, which comprises a list of input channel ends. A `ChannelInpuitList` is provided as one of the Groovy helper classes in the package `org.jcsp.groovy`. It makes for easier processing of collections of channels. There is a similar object for output channel ends called `ChannelOutputList`. The property `headings` {209} is a `List` of the same size as `inChannels`, though this is not checked, of the title to be placed at the top of each column of printed numbers. The property `delay` {210} is used to introduce a time delay between each line of printed output to make it easier to read as the output appears. The `delay` has a default value of 200 milliseconds and is of type `long` because the system clock returns times in that format. The default value will be used if the property is not assigned a new value when the process is constructed.

```
204    import org.jcsp.lang.*
205    import org.jcsp.groovy.*
206    import org.jcsp.plugNplay.ProcessRead

207    class GParPrint implements CSProcess {

208      def ChannelInputList inChannels
209      def List headings
210      def long delay = 200

211      void run() {
212        def inSize = inChannels.size()
213        def readerList = []
214        (0 ..< inSize).each { i ->
215                                readerList [i] = new ProcessRead ( inChannels[i] )
216                              }
217        def parRead = new PAR ( readerList )

218        if ( headings == null ) {
219          println "No headings provided"
220        }
221        else {
222          headings.each { print "\t${it}" }
223          println ()
224        }

225        def timer = new CSTimer()

226        while ( true) {
227          parRead.run()
228          readerList.each { pr -> print "\t" + pr.value.toString() }
229          println ()
230          if (delay != 0 ) {
231            timer.sleep ( delay)
232          }
233        }
234      }
235    }
```

Listing 3-15 The GParPrint Process Defintion

The number of `inChannels` in the `ChannelInputList` is obtained by applying the `size()` {212} method. The variable `readerList` is defined {264} as an empty list and will be used to build the list of `ProcessRead` processes that will be used to read from each of the `inChannels` in parallel. The `closure` {214-216} iterates over each element in the range `0` to `inSize-1` and constructs a `ProcessRead` process accessing the `i`'th element of `inChannels` and allocating the instance to the corresponding element of `readerList` {215}. A parallel is then constructed, `parRead`, using `PAR`, from `readerList` {217). The collection of processes is not executed at this time.

The heading for each column of output is now created {218-224}. If the value of `headings` is `null` {218} then a message indicating that no headings was provided is output {219}. Otherwise a heading is written, tab separated (`\t`) using the elements of the `List headings` by a closure that iterates {222} over each element of `headings`, using the `each` iterator method. The name `it` refers to the value returned by the iterator. It is assumed but not checked that the number of elements in headings is the same as that in `inChannels`.

A `timer` is now defined {225} of type `CSTimer` (see Chapter 9) that will be used to create the delay between each line of output. The main loop of the process can now commence {226-233}. The first requirement is to read the input values in parallel by executing `parRead` {227}. Once all the values have been read on all the input channels, in any order, then we can print the values to the console window. This is achieved by the use of a closure that iterates over `each` of the elements in `readerList` {228}. It is assumed that any object printed by this process will have the method `toString()` defined. The variable `pr` is assigned, in turn, each list element from which we extract the `value` field that can then be printed. If the value of `delay` is greater than zero then the `sleep` method is called on `timer`, which causes this process to stop execution, idle, for at least `delay` milliseconds {230-232}.

We can now use this process to print out all the intermediate values in the process network shown in Figure 3-10. This is simply achieved by inserting `GPCopy` processes into each connecting channel and sending one output to the next process and the other into the `GParPrint` process as shown in Figure 3-11. Arrays of channels are used to make naming easier as shown in Listing 3-16. The channels `connect` form links between the processes as a long chain or pipeline. The channels `outChans` provide the connection between the intermediate `GPCopy` processes and the final process to the `GParPrint` process.

The output from `GNumbers` is sent via `connect[0]` to the first instance of `GPCopy` which outputs the value in parallel to `connect[1]` and `outChans[0]`. Channel `connect[1]` then forms the input to `GIntegrate`, the output from which is communicated on `connect[2]` to the second instance of `GPCopy`. Channel `connect[3]` then sends the data stream to an instance of `GPairs`, the output of which is sent via `connect[4]` to an instance of `GPrefix`, which then finally sends the stream to `outChans[2]`. The `GPrefix` process has been inserted so that the tabular output is formatted correctly with a first line of zeros. Recall that `GPairs` consumes the first pair of numbers and only outputs a single number, hence we need to insert another number, 0, to form the tabular output correctly.

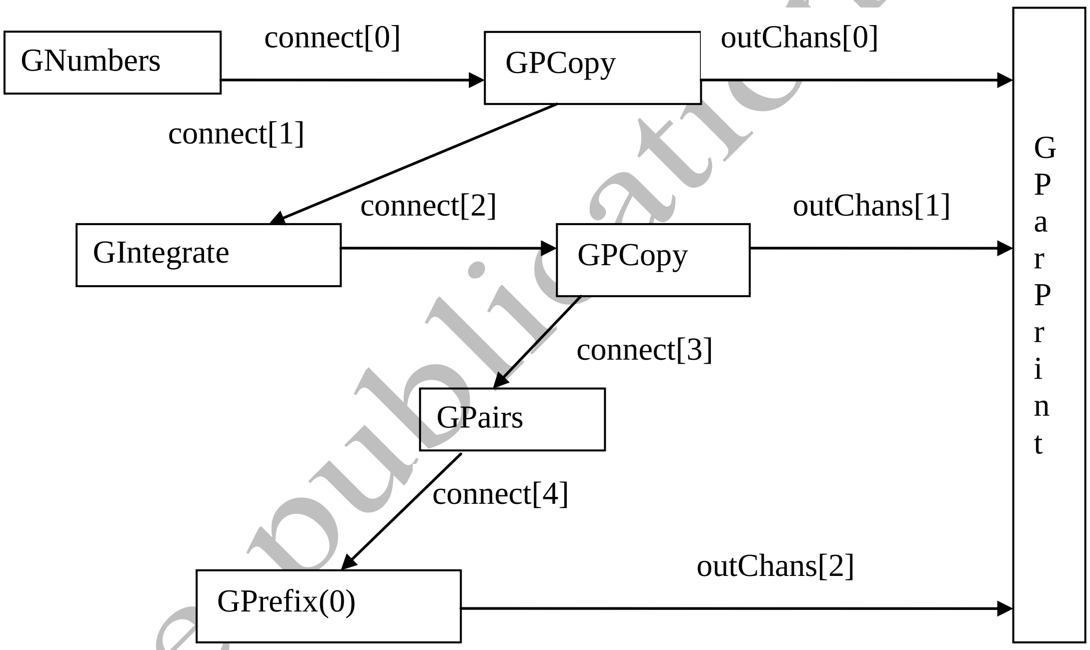

Figure 3-11 The Squares Network with Additional Printing

The arrays of channels are defined using an array constructor as shown in {238-239}. The list of inputs to `GParPrint` are created by means of the constructor for `ChannelInputList`, which takes a parameter of an array of channels and returns a list of channel input ends {240}. The list of `Strings` that make the `titles` of the columns is then defined {241}. The list of processes as shown in Figure 3-11 is then created connecting all the processes together {242-258}. Finally, the list of processes is invoked {259} and produces the output shown in Output 3-4.

```
n       int     sqr
0       0       0
1       1       1
2       3       4
3       6       9
4       10      16
5       15      25
6       21      36
7       28      49
8       36      64
9       45      81
10      55      100
11      66      121
12      78      144
13      91      169
14      105     196
15      120     225
16      136     256
```

Output 3-4 Table of Numbers Showing Intermediate Stages in the Calculation of Squares

```
import org.jcsp.lang.*
import org.jcsp.groovy.*

One2OneChannel [] connect = Channel.createOne2One(5)
One2OneChannel [] outChans = Channel.createOne2One(3)

def printList = new ChannelInputList ( outChans )

def titles = [ "n", "int", "sqr" ]

def testList =  [ new GNumbers   ( outChannel: connect[0].out() ),

                  new GPCopy    ( inChannel: connect[0].in(),
                                   outChannel0: connect[1].out(),
                                   outChannel1: outChans[0].out() ),

                  new GIntegrate ( inChannel: connect[1].in(),
                                   outChannel: connect[2].out() ),

                  new GPCopy     ( inChannel: connect[2].in(),
                                   outChannel0: connect[3].out(),
                                   outChannel1: outChans[1].out() ),

                  new GPairs     ( inChannel: connect[3].in(),
                                   outChannel: connect[4].out() ),

                  new GPrefix    ( prefixValue: 0,
                                   inChannel: connect[4].in(),
                                   outChannel: outChans[2].out() ),

                  new GParPrint  ( inChannels: printList,
                                   headings: titles )
                ]

new PAR ( testList ).run()
```

Listing 3-16 Script to Invoke the Process Network Shown in Figure 3-11

Consideration of the output shows that the numbers do appear in sequence in the column headed "n". The column headed "int" does contain the running sum or integration of the numbers. If we ignore the zero appearing in the first row of the column of squares headed "sqr", which was generated by the `GPrefix` process, then we see that there is indeed a list of the squares of the numbers in the first column.

3.10 Summary

We are now able to see why we can assert that this style of parallel processing has a compositional semantics. We know that each process is correct in its own right. By using them together, in a composition, we can go from a statement of what is required; generate a 0, generate a 1, then add the sequence up in pairs to a network that directly implements the requirement. We have also reused previously defined processes. This reuse and compositional capability means the system designer has to understand the operation of each of the processes in terms of the use of a process' input and output channels, so they can be correctly connected to each other. It is for this reason that the types of channels have been specified for class properties even though Groovy does not specifically require this to be done. In this simple case we have not specified the nature of the object that is communicated over the channels, as they are all of type `Integer`. In more complex systems the objects to be communicated should be documented as well.

Of more importance, is we have reused a number of processes, in relatively simple networks, to create a number of interesting results. We have also learnt that it is better to reuse existing processes wherever possible, rather than writing new processes, even if this means that we have to write another process. Parallel processing is not just a means of executing systems over a number of processors it also allows us to design systems more easily by composing existing processes into larger systems.

3.11 Exercises

1. Write a process that undoes the effect of `GIntegrate`. This can be achieved in two ways, first, by writing a `Minus` process that subtracts pairs of numbers read in parallel similar to `GPlus` or by implementing a `Negator` process and inserting it before a `GPlus` process. Implement both approaches and test them. Which is the more pleasing solution? Why?

1. Write a sequential version of `GPCopy`, called `GSCopy` that has the same properties as `GPCopy`. Make a copy of Listing 3-13 replacing `GPCopy` by your `GSCopy` and call it `GSPairsA`. Create another version, called `GSPairsB` in which the output channels `outChannel0` and `outChannel1` are assigned to the other actual channel, that is `a.out()` is assigned to `outChannel1` and `b.out()` is assigned to `outChannel1`. Take Listing 3-14 as the basis and replace `GPairs` by `GSPairsA` or `GSPairsB` and determine the effect of the change. Why does this happen? The accompanying web site contains the basis for this exercise apart from the body of `GSCopy`. Hint: read Section 4.7.2 that describes the operation of `GTail`.

2. Somewhat harder: Why was it considered easier to build `GParPrint` as a new process rather than using multiple instances of `GPrint` to output the table of results?

3. A `ChannelInputList` has a `read()` method that inputs from each channel in the channel list in parallel and returns a list, the same size as the `ChannelInputList` containing the object that has been read from each channel in the `ChannelInputList`. Modify the coding of Listing 3-15 to make use of this capability.