Communicating Process Architectures 2005
Jan Broenink, Herman Roebbers, Johan Sunter, Peter Welch, and David Wood (Eds.)
IOS Press, 2005

Groovy Parallel! A Return to the Spirit of occam?

Jon KERRIDGE, Ken BARCLAY and John SAVAGE
The School of Computing, Napier University, Edinburgh EH10 5DT
{j.kerridge, k.barclay, j.savage} @ napier.ac.uk

Abstract. For some years there has been much activity in developing CSP-like extensions to a number of common programming languages. In particular, a number of groups have looked at extensions to Java. Recent developments in the Java platform have resulted in groups proposing more expressive problem solving environments. Groovy is one of these developments. Four constructs are proposed that support the writing of parallel systems using the JCSP package. The use of these constructs is then demonstrated in a number of examples, both concurrent and parallel. A mechanism for writing XML descriptions of concurrent systems is described and it is shown how this is integrated into the Groovy environment. Finally conclusions are drawn relating to the use of the constructs, particularly in a teaching and learning environment.

Keywords. Groovy, JCSP, Parallel and Concurrent Systems, Teaching and Learning

Introduction

The occam programming language [1] provided a concise, simple and elegant means of describing computing systems comprising multiple processes running on one or more processors. Its theoretical foundations lay in the *Communicating Sequential Process* algebra of Hoare [2]. A practical realization of occam was the Inmos Transputer. With the demise of that technology the utility of occam as a generally available language was lost.

The Communicating Process Architecture community kept the underlying principles of occam alive by a number of developments such as Welch's JCSP package [3] and Hilderink's CTJ[4]. Both these developments captured the concept of CSP in a Java environment. The former is supported by an extensive package that also permits the creation of systems that operate over a TCP/IP network. The problem with the Java environment is that it requires a great deal of support code to create what is, in essence, a simple idea.

Groovy [5] is a new scripting language being developed for the Java platform. Groovy is compatible with Java at the bytecode level. This means that Groovy *is* Java. It has a Java friendly syntax that makes the Java APIs easier to use. As a scripting language it offers an ideal way in which to glue components. Groovy provides native syntactic support for many constructs such as lists, maps and regular expressions. It provides for dynamic typing which can immediately reduce the code bulk. The Groovy framework removes the heavy lifting otherwise found in Java.

Thus the goal of the activity reported in this paper was to create a number of simple constructs that permitted the construction of parallel systems more easily without the need for the somewhat heavyweight requirements imposed by Java. This was seen as particularly important when the concepts are being taught. By reducing the amount that has to be written, students may be able to grasp more easily the underlying principles.

1. The Spirit of Groovy

In August 2003 the Groovy project was initiated at codehaus [5], an open-source project repository focussed on practical Java applications. The main architects of the language are two consultants, James Strachan and Bob McWhirter. In its short life Groovy has stimulated a great deal of interest in the Java community. So much so that it is likely to be accepted as a standard language for the Java platform.

Groovy is a scripting language based on several languages including Java, Ruby, Python and Smalltalk. Although the Java programming language is a very good systems programming language, it is rather verbose and clumsy when used for systems integration. However, Groovy with a friendly Java-based syntax makes it much easier to use the Java Application Programming Interface. It is ideal for the rapid development of small to medium sized applications.

Groovy offers native syntax support for various abstractions. These and other language features make Groovy a viable alternative to Java. For example, the Java programmer wishing to construct a list of bank accounts would first have to create an object of the class `ArrayList`, then send it repeated `add` messages to populate it with `Account` objects. In Groovy, it is much easier:

```
accounts = [ new Account(number : 123, balance : 1200),
             new Account(number : 456, balance : 400)]
```

Here, the subscript brackets [and] denote a Groovy `List`. Observe also the construction of the `Account` objects. This is an example of a named property map. Each property of the `Account` object is named along with its initial value.

Maps (dictionaries) are also directly supported in Groovy. A `Map` is a collection of key/value pairs. A `Map` is presented as a comma-separated list of key : value pairs as in:

```
divisors = [4 : [2], 6 : [2, 3], 12 : [2, 3, 4, 6]]
```

This `Map` is keyed by an integer and the value is a `List` of integers that are divisors of the key.

Closures, in Groovy, are a powerful way of representing blocks of executable code. Since closures are objects they can be passed around as, for example, method parameters. Because closures are code blocks they can also be executed when required. Like methods, closures can be defined in terms of one or more parameters. One of the most common uses for closures is to process a collection. We can iterate across the elements of a collection and apply the closure to them. A simple parameterized closure is:

```
greeting = { name -> println "Hello ${name}" }
```

The code block identified by `greeting` can be executed with the call message as in:

```
greeting.call ("Jon")  // explicit call
greeting ("Ken")       // implicit call
```

Several `List` and `Map` methods accept closures as an actual parameter. This combination of closures and collections provides Groovy with some very neat solutions to common problems. The each method, for example, can be used to iterate across the elements of a collection and apply the closure, as in:

```
[1, 2, 3, 4].each { element -> print "${element}; " }
```

will print 1; 2; 3; 4;

```
["Ken" : 21, "John" : 22, "Jon" : 25].each { entry ->
    if(entry.value > 21) print "entry.key, "
}
```

will print

```
John, Jon,
```

2. The Groovy Parallel Constructs

Groovy constructs are required that follow explicit requirements of CSP-based systems. These are direct support for parallel, alternative and the construction of guards reflecting that Groovy is a list-based environment whereas JCSP is an array-based system [5].

2.1 The `PAR` Construct

The `PAR` construct is simply an extension of the existing JCSP `Parallel` class that accepts a list of processes. The class comprises a constructor that takes a list of processes (`processList`) and casts them as an array of `CSProcess` as required by JCSP.

```
class PAR extends Parallel {
  PAR(processList){
    super( processList.toArray(new CSProcess[0]) )
  }
}
```

2.2 The `ALT` construct

The `ALT` construct extends the existing JCSP `Alternative` class with a list of guards. The class comprises a constructor that takes a list of guards (`guardList`) and casts them as an array of `Guard` as required by the JCSP. The main advantage of this constructor in use is that the channels that form the guards of the `ALT` are passed to a process as a list of channel inputs and thus it is not necessary to create the `Guard` structure in the process definition. The list of guards can also include `CSTimer` and `Skip`.

```
class ALT extends Alternative {
  ALT (guardList) {
    super( guardList.toArray(new Guard[0]) )
  }
}
```

2.3 The `CHANNEL_INPUT_LIST` Construct

The `CHANNEL_INPUT_LIST` is used to create a list of channel input ends from an array of channels. This list can then be passed as a `guardList` to an `ALT`. This construct only needs to be used for channel arrays used between processes on a single processor. Channels that connect processes running on different processes (`NetChannels`) can be passed as a list without the need for this construct.

```
class CHANNEL_INPUT_LIST extends ArrayList{
  CHANNEL_INPUT_LIST(array) {
    super( Arrays.asList(Channel.getInputArray(array)) )
  }
}
```

2.4 The CHANNEL_OUTPUT_LIST Construct

The CHANNEL_OUTPUT_LIST is used to construct a list of channel output ends form an array of such channels and provides the converse capability to a CHANNEL_INPUT_LIST. It should be noted that all the channel output ends have to be accessed by the same process.

```
class CHANNEL_OUTPUT_LIST extends ArrayList{
  CHANNEL_OUTPUT_LIST(array) {
    super( Arrays.asList(Channel.getOutputArray(array)) )
  }
}
```

3. Using the Constructs

In this section we demonstrate the use of these constructs, first in a typical student learning example based upon the use of a number of sender processes having their outputs multiplexed into a single reading process. The second example is a little more complex and shows a system that runs over a network of workstations and provides the basic control for a tournament in which a number of players of different capabilities play the same game (draughts) against each other and this is then used in an evolutionary system to develop a better draughts player.

3.1 A Multiplexing System

3.1.1 The Send Process

The specification of the class SendProcess is brief and contains only the information required. This aids teaching and learning and also understanding the purpose of the process. The properties of the class are defined as cout and id (lines 2 and 3) without any type information. The property cout will be passed the channel used to output data from this process and id is an identifier for this process. The method run is then defined.

```
01  class SendProcess implements CSProcess {
02    cout  // the channel used to output the data stream
03    id    // the identifier of this process
04    void run() {
05      i = 0
06      1.upto(10) {          // loop 10 times
07        i = i + 1
08        cout.write(i + id)  // write the value of id + i to cout
09      }
10    }
11  }
```

There is no necessity for a constructor for the class or the setter and getter methods as these are all created automatically by the Groovy system. The run method simply loops 10 times outputting the value of id to which has been added the loop index variable i (lines 4 to 8). Thus the explanation of its operation simply focuses on the communication aspects of the process.

3.1.2 The Read Process

The ReadProcess is similarly brief and in this version extracts the SendProcess identification (s) and value (v) from the value that is sent to the ReadProcess. It should also be noted that types might be explicitly defined, as in the case of s (line 18), in order to

achieve the desired effect. It is assumed that identification values are expressed in thousands.

```
12  class ReadProcess implements CSProcess {
13    cin           // the input channel
14    void run() {
15      while (true) {
16        d = cin.read()                          // read from cin
17        v = d % 1000                            // v the value read
18        int s = d / 1000                        // from sender s
19        println "Read: ${v} from sender ${s}"  // print v and s
20      }
21    }
22  }
```

3.1.3 The `Plex` Process

The `Plex` process is a classic example of a multiplex process that alternates over its input channels (`cin`) and then reads a selected input, which is immediately written to the output channel (`cout`) (line 31). The input channels are passed as a list to the process and these are then passed to the `ALT` construct (line 27) to create the JCSP `Alternative`.

```
23  class Plex implements CSProcess {
24    cin       // channel input list
25    cout      // output channel onto which inputs are multiplexed
26    void run () {
27      alt = new ALT(cin)
28      running = true
29      while (running) {
30        index = alt.select ()
31        cout.write (cin[index].read())
32      }
33    }
34  }
```

3.1.4 Running the System on a Single Processor

Figure 1, shows a system comprising any number of `SendProcess`es together with a `Plex` and a `ReadProcess`.

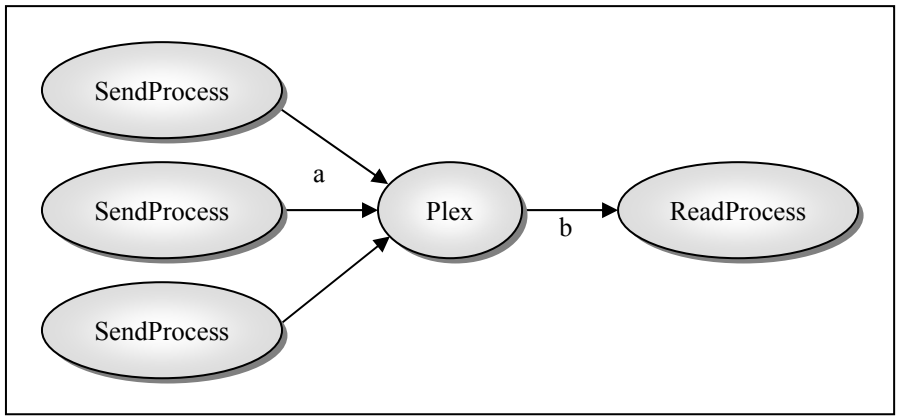

Figure 1. The Multiplex Process Structure

In a single processor invocation, five channels `a,` connect the `SendProcess`es to the `Plex` process and are declared using the normal call to the `Channel` class of JCSP (line 35). Similarly, the channel `b`, connects the `Plex` process to the `ReadProcess` (line 36). A `CHANNEL_INPUT_LIST` construct is used to create the list of channel inputs that will be passed to the `Plex` process and which will be ALTed over (line 37).

The Groovy `map` abstraction is used (line 38) to create `idMap` that relates the instance number of the `SendProcess` to the value that will be passed as its `id` property. A list (`sendList`) of `SendProcess`es is then created (lines 39-41) using the `collect` method on a list. The list comprises five instances of the `SendProcess` with the `cout` and `id` properties set to the values indicated, using a closure applied to each member of the set [0,1,2,3,4]. A `processList` is then created (lines 42-45) comprising the `sendList` plus instances of the `Plex` and `ReadProcess` that have their properties initialized as indicated. The `flatten()` method has to be applied because `sendList` is already a List that has to be removed for the PAR constructor to work. Finally a PAR construct is created (line 46) and run. In section 4 a formulation that removes the need for `flatten()` is presented.

```
35  a = Channel.createOne2One (5)
36  b = Channel.createOne2One ()
37  channelList = new CHANNEL_INPUT_LIST (a)
38  idMap = [0: 1000, 1: 2000, 2:3000, 3:4000, 4:5000]
39  sendList = [0,1,2,3,4].collect
40             {i->return new SendProcess ( cout:a[i].out(),
41                                          id:idMap[i]) }
42  processList = [ sendList,
43                  new Plex (cin : channelList, cout : b.out()),
44                  new ReadProcess (cin : b.in() )
45                ].flatten()
46  new PAR (processList).run()
```

3.1.5 Running the System in Parallel on a Network

To run the same system shown in Figure 1, on a network, with each process being run on a separate processor, a Main program for each process is required.

3.1.5.1 `SendMain`

`SendMain` is passed the numeric identifier (`sendId`) for this process (line 47) as the zero'th command line argument. A network node is then created (line 48) and connected to a default `CNSServer` process running on the network. From the `sendId`, a string is created that is the name of the channel that this `SendProcess` will output its data on and a `One2Net` channel is accordingly created (line 51). A list containing just one process is created (line 52) that is the invocation of the `SendProcess` with its properties initialized and this is passed to a `PAR` constructor to be run (line 53).

```
47  sendId = Integer.parseInt( args[0] )
48  Node.getInstance().init(new TCPIPNodeFactory ())
49  int sendInstance = sendId / 1000
50  channelInstance = sendInstance - 1
51  outChan = CNS.createOne2Net ( "A" + channelInstance)
52  pList = [ new SendProcess ( id : sendId, cout : outChan ) ]
53  new PAR(pList).run()
```

3.1.5.2 `PlexMain`

`PlexMain` is passed the number of `SendProcess`es as a command line argument (line 54), as there will be this number of input channels to the `Plex` process. These input channels are created as a list of `Net2One` channels (lines 57-59) having the same names as were created for each of the `SendProcess`es. As this is already a list there is no need to obtain the input ends of the channels, as this is implicit in the creation of `Net2One` channels. The `Plex outChan` is created as a `One2Net` channel with the name `B` (line 60) and the `Plex` process is then run in a similar manner as each of the `SendProcess`es (lines 61, 62).

```
54  inputs = Integer.parseInt( args[0] )
55  Node.getInstance().init(new TCPIPNodeFactory ())
56  inChans = []                        // an empty list of net channels
57  for (i in 0 ... inputs ) {
58    inChans << CNS.createNet2One ( "A" + i )  // append the channels
59  }
60  outChan = CNS.createOne2Net ( "B" )
61  pList = [ new Plex ( cin : inChans, cout : outChan ) ]
62  new PAR (pList).run()
```

3.1.5.3 ReadMain

`ReadMain` requires no command line arguments. It simply creates a network node (line 63), followed by a Net2One channel with the same name as was created for `PlexMain`'s output channel (line 64) and the `ReadProcess` is then invoked in the usual manner.

```
63  Node.getInstance().init(new TCPIPNodeFactory ())
64  inChan = CNS.createNet2One ( "B" )
65  pList = [ new ReadProcess ( cin : inChan ) ]
66  new PAR (pList).run()
```

3.1.6 Summary

In the single processor case, each process is interleaved on a single processor. In the multi-processor case each process is run on a separate processor and it is assumed that `CNSServer` [6] is executing somewhere on the network.

3.2 A `Tournament` Manager

The `Tournament` System, see Figure 2, is organized as a set of `Board` processes that each run a game in the tournament on a different processor. The `Board` processes receive information about the game they are to play from an `Organiser` process. The results from the `Board` processes are returned via a `ResultMux` process running on the same processor as the `Organiser` process. In order that the system operates in a Client-Server[6] mode each `Board` process is considered to be a client process and the combination of the `Organiser` and `ResultMux` processes is considered to be the Server.

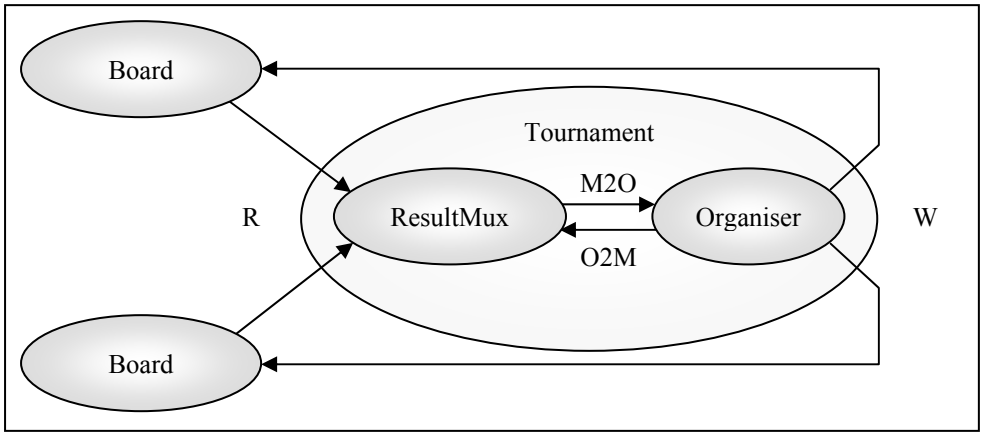

Figure 2 The Tournament System

The system requires that data be communicated as a set of `GameData` and `ResultData` objects. The system, as defined, cannot be executed on a single processor system as due account of the copying of network communicated objects, which have to implement

`Serializable`, is taken in the design. More importantly, the use of an internal channel between two processes has to be considered and a reply channel is utilized to overcome the fact that an object reference is passed between the `ResultMux` and `Organiser` processes.

3.2.1 The `Data` Objects

Two data objects are used within the system, `GameData` holds information concerning the player identities and the playing weights associated with each player. A `state` (line 72) property is used to indicate whether the object holds playing data or is being used to indicate the end of the `Tournament`.

```
67  class GameData implements Serializable {
68    p1          // id of player 1
69    p2          // id of player 2
70    w1          // list of weights for player 1
71    w2          // list of weights for player 2
72    state       // string containing data or end
73  }
```

The `ResultData` object is used to communicate results from the `Board` processes back to the `Organiser` process. The use of each property of the object is identified in the corresponding comments. The `board` on which the game is played is required (line 79) so the `Organiser` process can send another game to the `Board` process immediately. The `state` property (line 80) is used to indicate one of three states, namely; the board has been initialized waiting for a game, the object contains the results of a game and the tournament is finishing.

```
74  class ResultData implements Serializable {
75    p1          // player 1 identifier
76    p2          // player 2 identifier
77    result1V2   // result of game for p1 V p2
78    result2V1   // result of game for p2 V p1
79    board       // board used
80    state       // String containing init or result or end
81  }
```

3.2.2 The `Board` Process

The `Board` process is a client process and has been constructed so that an output to the `Organiser` in the form of a `result.write()` (lines 96, 103, 119) communication is always followed immediately by a `work.read()` (line 98). The initialization code with its output is immediately followed, in the main loop, by the required input operation. The main loop comprises two sections of an if-statement, which finish with either the outputting of a result or a termination message. The latter does not need to receive an input from the `Organiser` process because the `Board` process will itself have been terminated. In the normal case, the outputting of a result at the end of the loop is immediately followed by an input at the start of the loop. These lines (96, 98, 103, 119) have been highlighted in the code listing. A consequence of using this design approach is that only one `ResultData` and one `GameData` object is required thereby minimizing the use of the very expensive `new` operator.

The most interesting aspect of the code is that the access to the properties of the data classes is simply made using the dot notation. This results from Groovy automatically generating the setters, getters and class constructors required. This has the immediate benefit of making the code more accessible so that key points such as the structure of client and server processes is more obvious.

```
82  class Board implements CSProcess {
83
84    bId             // the id for this Board process
85    result          // One2One channel connecting the Board to the ResultMux
86    work            // One2One channel used to send work to this Board
87
88    void run() {
89      println "Board ${bId} has started"
90      tim = new CSTimer()                   // used to simulate game time
91      gameData = new GameData()             // the weights and player ids
92      resultData = new ResultData()         // the result of this game
93      resultData.state = "init"
94      resultData.board = bId
95      running = true
96      result.write(resultData)              // send init to Organiser
97      while (running) {
98        gameData = work.read()              // always follows a result.write
99        if ( gameData.state == "end" ) {    // end of processing
100         println "Board ${bId} has terminated"
101         running = false
102         resultData.state = "end"
103         result.write(resultData)          // send termination to ResultMux
104       }
105       else {
106         // run the game twice with P1 v P2 and then P2 v P1
107         // simulated by a timeout
108         tim.after ( tim.read() + 100 + gameData.p2 )
109         println "Board ${bId} playing games for
110                  ${gameData.p1} and ${gameData.p2}"
111         outcome1V2 = bId       // return the bId of the board playing game
112         outcome2V1 = -bId      // instead of the actual outcomes
113         resultData.state = "result"
114         resultData.p1 = gameData.p1
115         resultData.p2 = gameData.p2
116         resultData.board = bId
117         resultData.result1V2 = outcome1V2
118         resultData.result2V1 = outcome2V1
119         result.write(resultData)          // send result to ResultMux
120       }
121     } } }
```

3.2.3 The `ResultMux` Process

This process forms part of the tournament system and is used to multiplex results from the `Board` processes to the `Organiser`. The `ResultMux` process runs on the same processor as the `Organiser` and thus access to any data objects by both processes have to be carefully managed. If this is not done then there is a chance that one process may overwrite data that has already been communicated to the other process because only an object reference is passed during such communications. In this case, the `resultData` object is read into in the `ResultMux` process and manipulated within `Organiser`. Yet again the desire is to reduce the number of `new` operations that are undertaken. `new` is both expensive and also leads to the repeated invocation of the Java garbage collector. In the version presented here only one instance of a `ResultData` object is created outside the main loop of the process. In addition, no `new` operation exists within the loop (lines 129-144).

The only other problem to be overcome is that of terminating the `ResultMux` process. One of the properties (`boards`) of the process is the number of parallel `Board` processes invoked by the system. When a `Board` process receives a `GameData` object that has its `state` set to "`end`" it communicates this to the `ResultMux` process as well. Once the `ResultMux` process has received the required number of such messages it can then terminate itself (lines 137-140).

The other aspect of note is that the property `resultsIn` is a list of network channels and that these can be used as a parameter to the `ALT` construct without any modification because `ALT` (line 132) is expecting a list of input channel ends, which is precisely the type of a Net2One channel, see 3.2.6. Any `ResultData` that is read in on the `resultsIn` channels is then immediately written to the `resultOut` channel (line 143).

The use of the `reply` property will be explained in the next section.

```
122 class ResultMux  implements CSProcess {
123   boards      // number of boards; used for process termination
124   resultOut   // output channel from  Mux to Organiser
125   reply       // channel indicating result processed by Organiser
126   resultsIn   // list of result channels from each of the boards
127
128   void run () {
129     resultData = new ResultData()   // holds data from boards
130     endCount = 0
131     println "ResultMux has started"
132     alt = new ALT (resultsIn)
133     running = true
134     while (running) {
135       index = alt.select()
136       resultData = resultsIn[index].read()
137       if ( resultData.state == "end" ) {
138         endCount = endCount + 1
139         if ( endCount == boards ) {
140           running = false
141         }
142       } else {
143         resultOut.write(resultData)
144         b = reply.read()
145       }
146     } } }
```

3.2.4 The `Organiser` Process

This is the most complex process but it breaks down into a number of distinct sections that facilitate its explanation. Yet again the use of the `new` operation has been limited to those structures that are required and none are contained within the main loop of the process. The `outcomes` structure is a list of lists that will contain the result of each game. The access mechanism is similar to that of array access but Groovy permits other styles of access that are more list oriented. Initially, each element of the structure is set to a sentinel value of 100 (lines 159-166). The result of each pair of games, p_i plays p_j and p_j plays p_i for all i <>j, is recorded in the `outcomes` structure such that p_i v p_j is stored in the upper triangle of `outcomes` and p_j v p_i in the lower part. Games such as draughts and chess have different outcomes for the same players depending upon which is white or black and hence is the starting player.

The main loop has been organized so that the `Organiser` receives a result from the `ResultMux`. Saving the game's results in the `outcomes` structure and then sending another game to the now idle `Board` process achieves this (lines 171-178). However, before another game is sent to the `Board` process a reply (line 178) is sent to the `ResultMux` process to indicate the `ResultData` has been processed. The `resultData` object is passed as a value from the `ResultMux` to the `Organiser`, which is an object reference. JCSP requires that once a process has written an object it should not then access that object until it is safe to do so. Thus once the `outcomes` structure has been updated the object is not required and hence the reply can be sent to the `ResultMux` process immediately. This happens on two occasions, first when the `resultData` contains the state "`init`" (line 180) and more commonly when a result is returned and the state is "`result`" (line 178).

```
147 class Organiser  implements CSProcess {
148   boards          // the number of boards that are being used in parallel
149   players         // number of players
150   work            // channels on which work is sent to boards
151   result          // channel on which results received from ResultMux
152   reply           // reply to resultMux from Organiser
153
154   void run () {
155     resultData = new ResultData()       // create the data structures
156     gameData = new GameData()
157     println "Organiser has started"
158     // set up the outcomes
159     outcomes = [ ]
160     for ( r in 0 ..< players ) {        // cycle through the rows
161       row = [ ]                         // 0 ..< n gives 0 to n - 1
162       for ( c in 0 ..< players ) {      // cycle through the columns
163         row << 100                      // 100 acts as sentinel
164       }
165       outcomes << row
166     }
167     // the main loop
168     for ( r in 0 ..< players) {
169       c = r + 1
170       for ( c in 0 ..< players) {
171         resultData = result.read()      // an object reference not a copy
172         b = resultData.board
173         if ( resultData.state == "result" ) {
174           p1 = resultData.p1
175           p2 = resultData.p2
176           outcomes [ p1 ] [ p2 ] = resultData.result1V2
177           outcomes [ p2 ] [ p1 ] = resultData.result2V1
178           reply.write(true)             // outcomes processed
179         } else {
180           reply.write(true)             // init received
181         }
182         // send the game [r,c] to Board process b
183         gameData.p1 = r
184         gameData.p2 = c
185         gameData.state = "data"
186         // set w1 to the weights for p1
187         // set w2 to the weights for p2
188         work[b].write(gameData)
189       }
190     }
191     // now terminate the Board processes
192     println "Organiser:  Started termination process"
193     gameData.state = "end"
194     for ( i in 0 ... boards) {
195       resultData = result.read()
196       bd = resultData.board
197       p1 = resultData.p1
198       p2 = resultData.p2
199       outcomes [ p1 ] [ p2 ] = resultData.result1V2
200       outcomes [ p2 ] [ p1 ] = resultData.result2V1
201       reply.write(true)
202       work[bd].write(gameData)
203     }
204     println"Organiser: Outcomes are:"
205     for ( r in 0 ... players ) {
206       for ( c in 0 ... players ) {
207         print "[${r},${c}]:${outcomes[r][c]}; "
208       }
209       println " "
210     }
211     println"Organiser: Tournament has finished"
212   }
213 }
```

Initially, the loop will receive as many "`init`" messages as there are `Board` processes. Thus once all the games have been sent to the `Board` processes, each of the `Board` processes will still be processing a game. Hence, another loop has to be used to input the last game result from each of these processes (lines 194-203). In this case the `gameData` that is output contains the state "`end`" and this will cause the `Board` process that receives it to terminate but not before it has also sent the message on to the `ResultMux` process. Finally, the `outcomes` can be printed (lines 204-211) or in the real tournament system evaluated to determine the best players so that they can be mutated in an evolutionary development scheme.

3.2.5 Invoking a `Board` Process

Each `Board` process has to be invoked on its own processor. The network channels are created using CNS static methods (lines 216, 217). It is vital that the channel names used in one process invocation are the same as the corresponding channel in another processor.

```
214 Node.getInstance().init(new TCPIPNodeFactory ());
215 boardId = Integer.parseInt(args[0])         //the number of this Board
216 w = CNS.createNet2One("W" + boardId)        // the Net2One work channel
217 r = CNS.createOne2Net("R" + boardId)        // the One2Net result channel
218 println " Board ${boardId} has created its Net channels "
219 pList = [ new Board ( bId:boardId , result:r , work:w ) ]
220 new PAR (pList).run()
```

3.2.6 Invoking the `Tournament`

This code is similar expect that list of network channels are created by appending channels of the correct type to list structures (lines 224-230). Two internal channels between `ResultMux` and `Organiser` are created, `M2O` and `O2M` (lines 231, 232) and these are used to implement the `resultOut` and `reply` connections respectively between these processes. An advantage of the Groovy approach to constructors is that the constructor identifies each property by name, rather than the order of arguments to a constructor call specifying the order of the properties. It also increases the readability of the resulting code.

```
221 Node.getInstance().init(new TCPIPNodeFactory ());
222 nPlayers = Integer.parseInt(args[0])        // the number of players
223 nBoards = Integer.parseInt(args[1])         // the number of boards
224 w = []  // the list of One2Net work channels
225 r = []  // the list of Net2One result channels
226 for ( i in 0 ..< nBoards) {
227   i = i+1
228   w << CNS.createOne2Net("W" + i)
229   r << CNS.createNet2One("R" + i)
230 }
231 M2O = Channel.createOne2One()
232 O2M = Channel.createOne2One()
233 pList = [ new Organiser ( boards:nBoards , players:nPlayers ,
234                           work:w , result: M2O.in(),
235                           reply: O2M.out() ),
236           new ResultMux ( boards:nBoards , resultOut:M2O.out(),
237                           resultsIn:r, reply: O2M.in() ) ]
238 new PAR ( pList) .run()
```

4. The XML Specification of Systems

Groovy includes tree-based *builders* that can be sub-classed to produce a variety of tree-structured object representations. These specialized builders can then be used to represent, for example, XML markup or GUI user interfaces. Whichever kind of builder object is used, the Groovy markup syntax is always the same. This gives Groovy native syntactic support for such constructs.

The following lines, 239 to 248, demonstrate how we might generate some XML [7] to represent a book with its author, title, etc. The non-existent method call `Author("Ken Barclay")` delivers the `<Author>Ken Barclay</Author>` element, while the method call `ISBN(number : "1234567890")` produces the empty XML element `<ISBN number= "1234567890"/>`.

```
239 //  Create a builder
240 mB = new MarkupBuilder()
241
242 //  Compose the builder
243 bk = mB.Book() {                    // <Book>
244     Author("Ken Barclay")           // <Author>Ken Barclay</Author>
245     Title("Groovy")                 // <Title>Groovy</Title>
246     Publisher("Elsevier")           // <Publisher>Elsevier</Publisher>
247     ISBN(number : "1234567890")     // <ISBN number="1234567890"/>
248                                     // </Book>
```

It is also important to recognize that since all this is native Groovy syntax being used to represent any arbitrarily nested markup, then we can also mix in any other Groovy constructs such as variables, control flow such as looping and branching, or true method calls.

In keeping with the spirit of Groovy, manipulating XML structures is made particularly easy. Associated with XML structures is the need to navigate through the content and extract various items. Having, say, parsed a data file of XML then traversing its structures is directly supported in Groovy with XPath-like [7] expressions. For example, a data file comprising a set of Book elements might be structured as:

```
249 <Library>
250     <Book> … </Book>
251     <Book> … </Book>
252     <Book> … </Book>
253     …
254 </Library>
```

If the variable `doc` represents the root for this XML document, then the navigation expression `doc.Book[0].Title[0]` obtains the first `Title` for the first `Book`. Equally, `doc.Book` delivers a `List` that represents all the `Book` elements in the `Library`. With a suitable iterator we immediately have the code to print the title of every book in the library:

```
255 parser = new XmlParser()
256 doc = parser.parse("library.xml")
257
258 doc.Book.each { bk ->
259     println "${bk.Title[0].text()}"
260 }
```

The ease with which Groovy can manipulate XML structures encourages us the consider representing JCSP networks as XML markup. Groovy can then manipulate that information, configure the processes and channels, and then execute the model. For

example, we might arrive at the following markup (lines 261-274) for the classical producer–consumer system built from the `SendProcess` and the `ReadProcess` described in 3.1.1 and 3.1.2. The libraries to be imported are specified on lines 262 and 263.

```
261 <csp-network>
262   <include name="com.quickstone.jcsp.lang.*"/>
263   <include name="uk.ac.napier.groovy.parallel.*"/>
264   <channel name="chan" class="Channel" type="createOne2One"/>
265     <processlist>
266         <process class="SendProcess">
267             <arg name="cout" value="chan.out()"/>
268             <arg name="id"   value="1000"/>
269         </process>
270         <process class="ReadProcess">
271             <arg name="cin" value="chan.in()"/>
272         </process>
273     </processlist>
274 </csp-network>
```

To ensure the consistency of the information contained in these network configurations we could define an XML schema [7] for this purpose. A richer schema defines how nested structures could be described. From the preceding example we also permit a recursive definition whereby a simple `<process>` may itself be another `<processlist>`. Hence we can define the XML for the plexing system described in 3.1.4 by the following.

```
275 <csp-network>
276     <include name="com.quickstone.jcsp.lang.*"/>
277     <include name="uk.ac.napier.groovy.parallel.*"/>
278     <channel name="a" class="Channel" type="createOne2One" size="5"/>
279     <channel name="b" class="Channel" type="createOne2One"/>
280     <channelInputList name="channelList" source="a"/>
281     <processlist>
282         <processlist>
283             <process class="SendProcess">
284                 <arg name="cout"    value="a[0].out()"/>
285                 <arg name="id"            value="1000"/>
286             </process>
287             <process class="SendProcess">
288                 <arg name="cout"    value="a[1].out()"/>
289                 <arg name="id"            value="2000"/>
290             </process>
291             <process class="SendProcess">
292                 <arg name="cout"    value="a[2].out()"/>
293                 <arg name="id"            value="3000"/>
294             </process>
295             <process class="SendProcess">
296                 <arg name="cout"    value="a[3].out()"/>
297                 <arg name="id"            value="4000"/>
298             </process>
299             <process class="SendProcess">
300                 <arg name="cout"    value="a[4].out()"/>
301                 <arg name="id"            value="5000"/>
302             </process>
303         </processlist>
304         <process class="Plex">
305             <arg name="cout"        value="b.out()"/>
306             <arg name="cin"         value="channelList"/>
307         </process>
308         <process class="ReadProcess">
309             <arg name="cin"         value="b.in()"/>
310         </process>
311     </processlist>
312 </csp-network>
313
```

By inspection we can see that the XML presented in lines 275 to 312 capture the Groovy specification of the system given in lines 35 to 46. The main difference is that the list of `SendProcesses` generated in lines 39 to 41 has been explicitly defined as a sequence of `SendProcess` definitions. A Groovy program can parse this XML and the system will then be invoked automatically on a single processor.

The automatically generated output from the above XML script is shown in lines 314 to 330. As can be seen it generates two `PAR` constructs nested one in the other. The internal one contains the list of `SendProcesses` that are included within the one running the `Plex` and `ReadProcess` processes. Lines 314 and 315 show the jar files that have to be imported. The Groovy Parallel constructs described in section 2 have been placed in a jar file, emphasizing that Groovy is just Java.

```
314 import com.quickstone.jcsp.lang.*
315 import uk.ac.napier.groovy.parallel.*
316 a = Channel.createOne2One(5)
317 b = Channel.createOne2One()
318 channelList = new CHANNEL_INPUT_LIST(a)
319 new PAR([
320     new PAR([
321         new SendProcess(cout : a[0].out(), id : 1000),
322         new SendProcess(cout : a[1].out(), id : 2000),
323         new SendProcess(cout : a[2].out(), id : 3000),
324         new SendProcess(cout : a[3].out(), id : 4000),
325         new SendProcess(cout : a[4].out(), id : 5000)
326     ]),
327     new Plex(cout : b.out(), cin : channelList),
328     new ReadProcess(cin : b.in())
329 ])
330 .run()
```

5. Conclusions and Future Work

The paper has shown that it is possible to create problem solutions in a clear and accessible manner such that the essence of the CSP-style primitives and operations is more easily understood. A special lecture was given to a set of students who were being taught Groovy as an optional module in their second year. This lecture covered the concepts of CSP and their implementation in Groovy. There was consensus that the approach had worked and that students were able to assimilate the ideas. This does however need to be tested further in a more formal setting.

Currently, Groovy uses dynamic binding and it can be argued that this is not appropriate for a proper software engineering language. It would only need for this checking to be done at compile time, say by a switch, and we could more robustly design, implement and test systems.

Work is being undertaken to develop a diagramming tool that outputs the XML required by the system builder. This would mean that the whole system could be seamlessly incorporated into existing design and development tools such as ROME [8]. This could be extended to develop techniques for distributing a parallel system over a network of workstations or a Beowulf cluster.

Further consideration could also be given to the XML specifications. An XML vocabulary might be developed that is richer than that presented. Such a vocabulary might provide a compact way to express for example, the channels used as inputs to processes where they become the `Guards` of an `ALT` construct.

Can we answer the question posed by the title of this paper in the affirmative? We suggest that sufficient evidence has been presented and that this provides a real way forward for promoting the design of systems involving concurrent and parallel components.

Acknowledgements

A colleague, Ken Chisholm, provided the requirement for the draughts tournament. The helpful comments of the referees were gratefully accepted.

References

[1] Inmos Ltd, *occam2 Programming Reference Manual*, Prentice-Hall, 1988.
[2] C.A.R. Hoare, *Communicating Sequential Processes.* New Jersey: Prentice-Hall, 1985; available electronically from `http://www.usingcsp.com/cspbook.pdf`.
[3] P.H. Welch, *Process Oriented Design for Java – Concurrency for All*, `http://www.cs.kent.ac.uk/projects/ofa/jcsp/jcsp.ppt`, web site accessed 4/5/2005.
[4] G. Hilderink, A. Bakkers and J. Broenink, *A Distributed Real-Time java System Based on CSP*, The Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, ISORC 2000, Newport Beach, California, pp.400-407, March 15-17, 2000.
[5] Groovy Developer's Web Site, accessed 4/5/2005, `groovy.codehaus.org`.
[6] Quickstone Ltd, web site accessed 4/5/2005, `www.quickstone.com`.
[7] `http://www.w3.org/TR/REC-xml/`; `http://www.w3.org/TR/xpath`.
[8] K. Barclay and J. Savage, *Object Oriented Design with UML and Java*, Elsevier 2004; supporting tool available from `http://www.dcs.napier.ac.uk/~kab/jeRome/jeRome.html`.