14 Barriers and Buckets: Hand-Eye Co-ordination Test

In this chapter three shared memory synchronisation techniques are combined to provide control of a highly dynamic environment. A `Barrier` provides a means whereby a known number of processes collectively control their operation so they all wait at the barrier until all of them have synchronised with the barrier at which time they are all released to run in parallel. An `AltingBarrier` is a specialisation of the `Barrier` that allows it to act also as a guard in an Alternative. Finally, a `Bucket` provides a flexible refinement of a barrier. Typically, there will be a collection of `Bucket`s into which processes are placed depending upon some criterion. Another process then, subsequently, causes a `Bucket` to flush all its processes so they are executed in parallel. These processes will in due course, become idle, whereupon they place themselves in other buckets. The next `Bucket` in sequence is then flushed and so the cycle is repeated. `Bucket`s can be used to control discrete event simulations in a very simple manner [HICSS]. The process that undertakes the flushing of the buckets must not be one of the processes that can reside in a `Bucket`.

The aim of this example is to present a user with a randomly chosen set of targets that each appear for a different random time. During the time the targets are available the user clicks the mouse over each of the targets in an attempt to hit as many of the targets as possible. The display includes information of how many targets have been hit and the total number of targets that have been displayed. The targets are represented by different coloured squares on a black background and a hit target is coloured white. A target that is not 'hit' before its self determined random time has elapsed is coloured grey. There is a gap between the end of one set of targets and the display of the next set during which time the screen is made all black. The minimum time for which a target is displayed is set by the user; obviously the longer this time the easier it is to hit the targets. Targets will be available for a period between the shortest time and twice that time. Figure 14-1 shows the screen, at the point when six targets have been displayed, and none have yet been hit. The system has displayed a total of 88 targets of which 15 targets have been hit. The minimum target delay was 900 milliseconds. It can be deduced there are 16 targets in a 4 x 4 matrix.

The solution presumes that each target is managed by its own process and that it is these processes that are held in a `Bucket` until it is the turn of that `Bucket` to be flushed. When a target is enabled it displays itself until either it is 'hit' by a mouse-click, in which case it turns white, or the time for which it appears elapses and it is coloured grey. It is obvious that each of these target processes will finish at a different time and because the number of targets is not predetermined a barrier is used to establish when all the enabled target processes have finished. After this, the target process determines into which bucket it is going fall and thereby remains inactive until that bucket is flushed. The other processes used in the solution are shown in Figure 14-2.

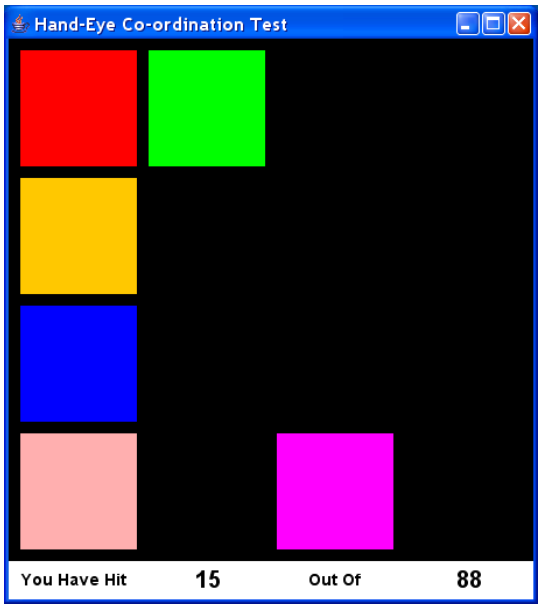

Figure 14-1 The Screen for the Hand-Eye Co-ordination Test

The system comprises a number of distinct phases each of which is controlled by its own barrier, which depending on the context is either a simple `Barrier` or an `AltingBarrier`.

Figure 14-2 shows the system at the point where it is about to synchronise on the `setUpBarrier`. During this setup phase there are no channel communications but the processes that synchronise on `setUpBarrier` either have to initialise themselves in some manner or must not progress beyond a certain point to ensure the system will not get out of step with itself. The setup phase only occurs once when the system is initially executed. The processes that are not part of the `setUpBarrier` cannot make any progress because they are dependent on other barriers or communications with processes that synchronise on the `setUpBarrier`.

The `BarrierManager` is a process that is used to manage phase synchronisations and as such will be seen in subsequent figures to be part of a number of other barriers. For ease of description the structure of each phase will show only the relevant barrier and channels that are operative at that time. The separation into these distinct phases also makes it easier to analyse the system from the point of view of its client-server architecture, thereby enabling deadlock and livelock analysis.

The `TargetFlusher` and `TargetProcess` processes are the only processes that can manipulate the array of `Buckets`, which are not shown on the diagram. The `TargetProcesses` are able to identify which `Bucket` they are going to enter when they stop running. `TargetFlusher` is the only process that can cause the execution of the processes contained with a `Bucket`. It is presumed that the cycle of a `TargetProcess` is to wait until it is flushed from a `Bucket`; it then runs until it determines, itself, that it has ceased to run at which point it causes itself to `fallInto` a `Bucket`, which it also determines.

The `DisplayController` process initialises the display window to black. It also initialises, to zero, the information contained in the display widow as to the number of hits that have occurred and the total number of targets that have been displayed.

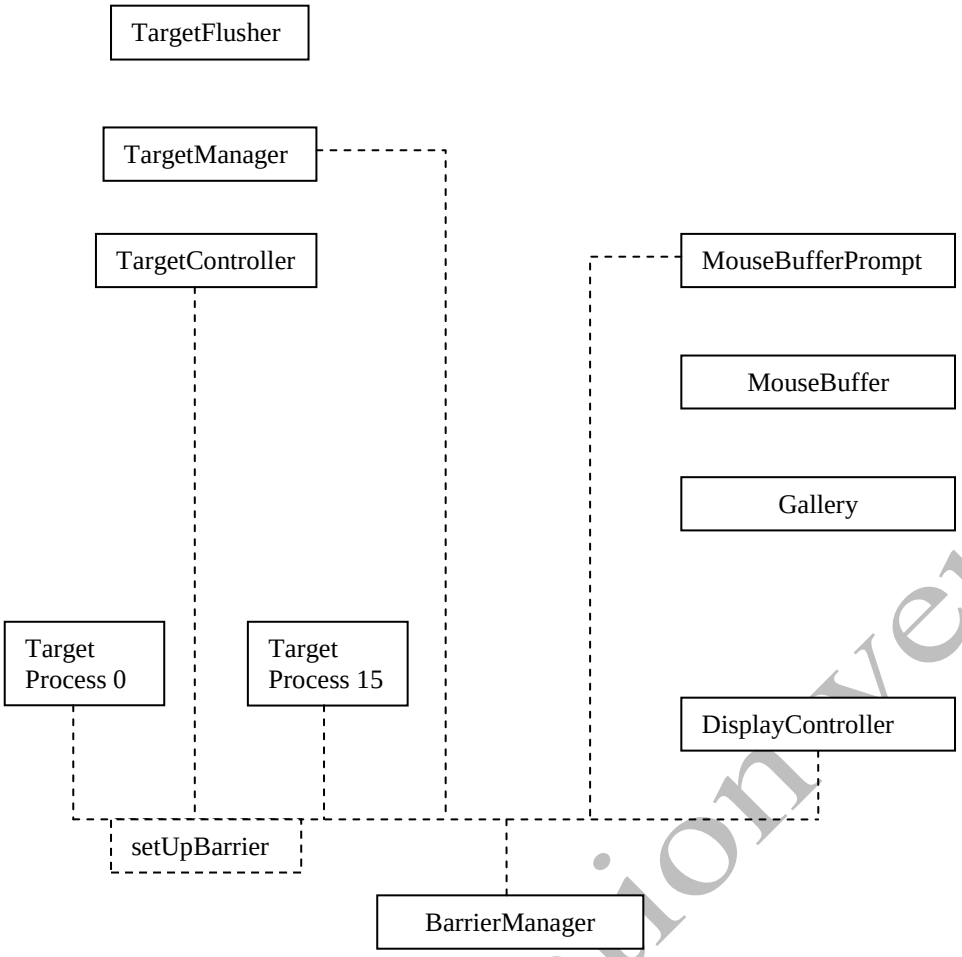

Figure 14-2 System At Setup Barrier Synchronisation

Figure 14-3 shows the system at the `initBarrier` synchronisation, which is the point at which those targets that are executing have initialised themselves and the associated display window is showing the targets. Prior to the `initBarrier` the only process that can execute is `TargetController`. The `TargetController` requests the `TargetManager` to flush the next `Bucket`; a request that is passed onto the `TargetFlusher` process. The `TargetFlusher` accesses the `Bucket`s in sequence until it finds a non-empty one. It then initialises the `initBarrier` with the number of `TargetProcesse`s. It returns this number to the `TargetManager` and then flushes the `TargetProcesse`s, which start running. The `TargetManager` then determines which of the `TargetProcesse`s has been started by waiting for a communication from each of them informing it of the identity of the running targets. These identities are then formed into a `List`, which is then communicated to both the `TargetController` and `DisplayController` processes.

The TargetController can now construct a `ChannelOutputList` that will be subsequently used to communicate the location where mouse clicks occur to each of the `TargetProcesse`s. Similarly, the `DisplayController` can modify the display window to show the running targets.

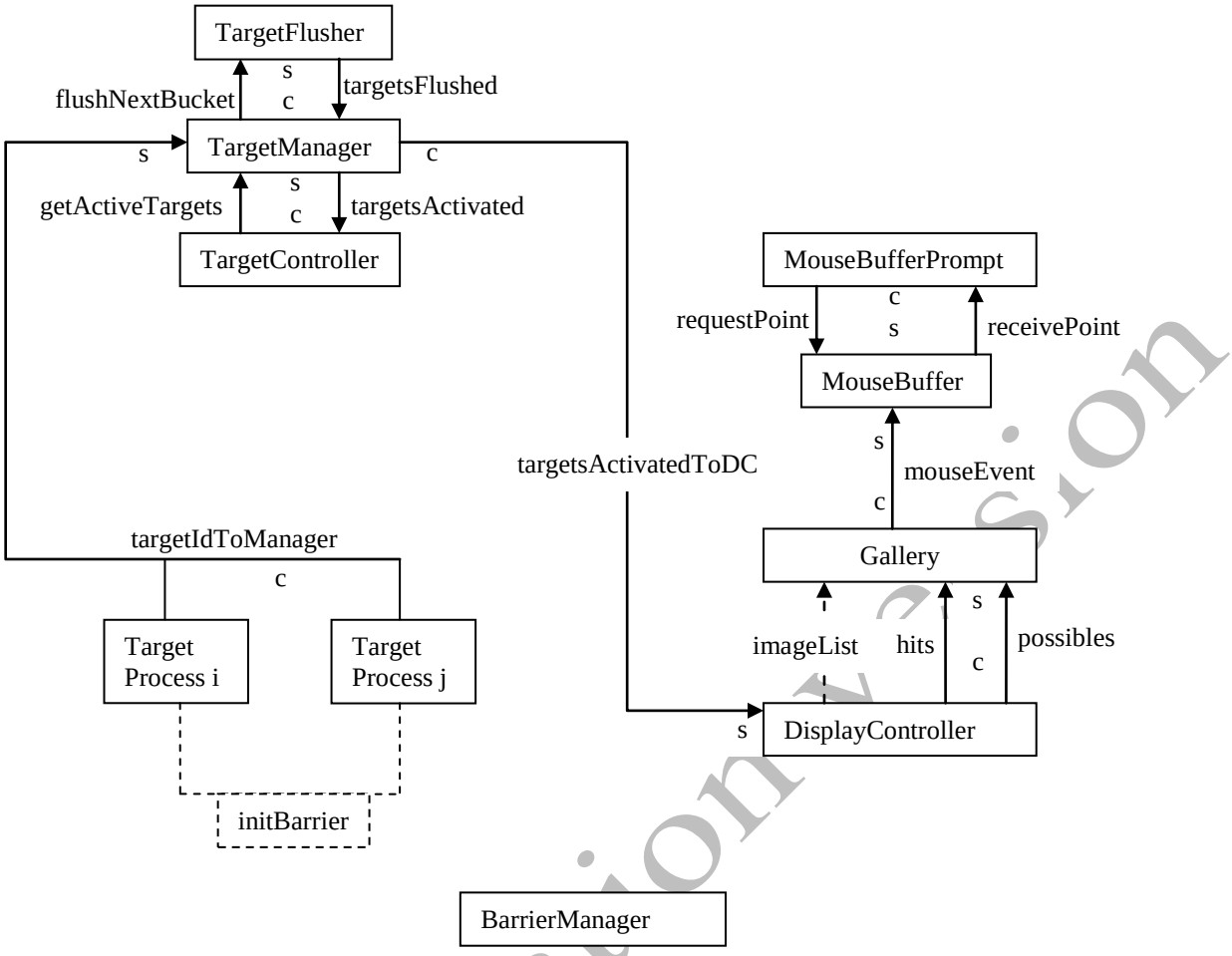

Figure 14-3 System At the Initialise Barrier Synchronisation

The `MouseBufferPrompt` and `MouseBuffer` have a design similar to that used previously in the manipulation of a queue (Chapter 6.2) and event handling (Chapter 11.2). `MouseBuffer` only accepts a request from `MouseBufferPrompt` when it has already received an event on its `mouseEvent` channel. The `Gallery` process is responsible both for the `ActiveCanvas` upon which the targets are displayed and the detection and communication of mouse click events. At this stage the `MouseBufferPrompt` process has no channel on which it can output points but that is not required until the system progresses to the next, `goBarrier` phase.

The `goBarrier` is simply required to ensure that all the running `TargetProcesses`, the `TargetController` and `DisplayController` have reached a state whereby the system can start execution from a known state. As such this phase does not require any channel communication as shown in Figure 14-4. Once these processes have synchronised the system enters the normal running state of the system with some of the `TargetProcesses` executing.

Each of the `Barriers` used so far are of the simple variety because the number of processes that require synchronising can be predetermined and there is no need for any of these `Barriers` to interact with a possible communication or timer in an alternative. The communications are all required to have completed before the processes can reach the synchronisation point. The remaining `Barriers` are of the `AltingBarrier` variety because the requirement to synchronise can happen at the same time as a timer alarm or communication occurs.

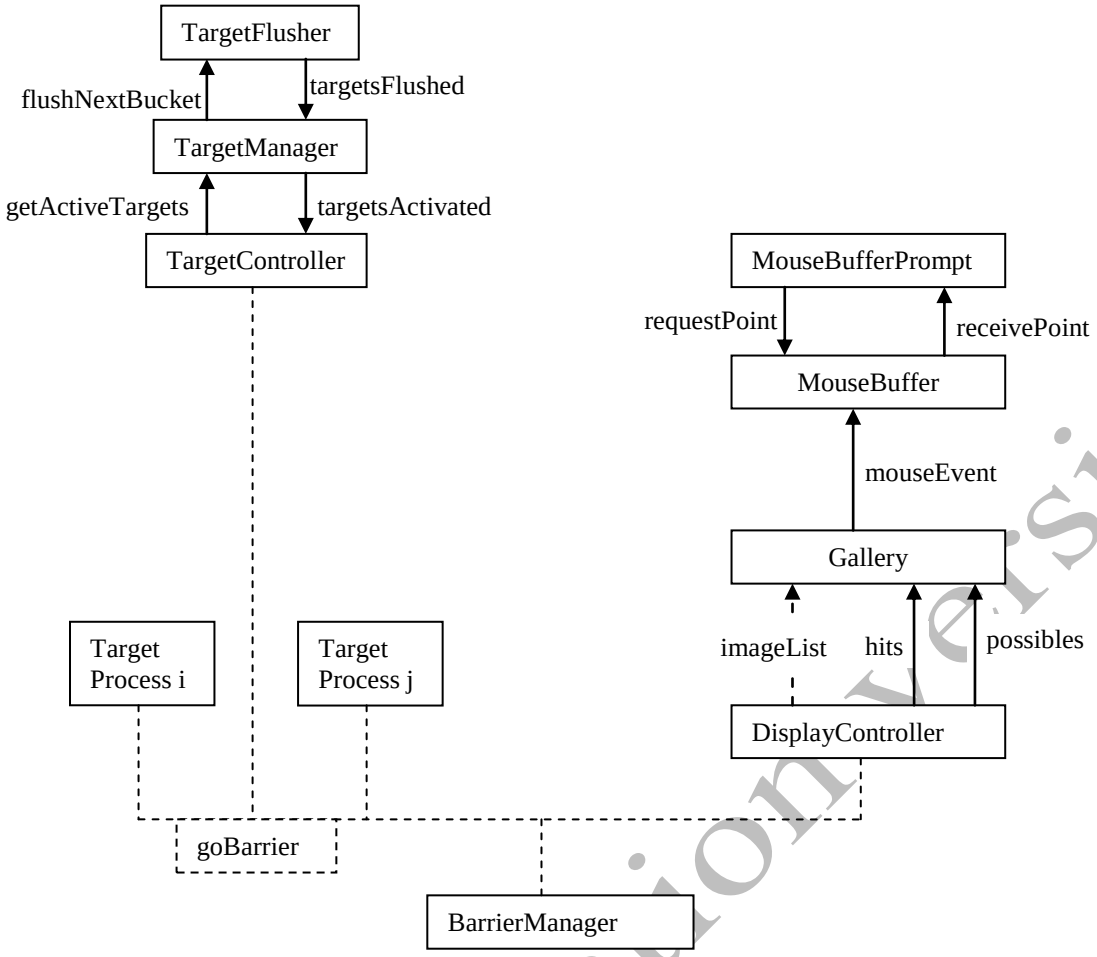

Figure 14-4 System At the Go Barrier Synchronisation

Figure 14-5 shows the system structure when the TargetProcesses are waiting for mouse clicks to determine whether or not they have been hit. The figure also shows the client-server analysis appropriate to this phase of the system's operation.

Initial, cursory inspection, would seem to suggest that there a client-server loop has been created. However, it can be seen that the `MouseBuffer` is a pure server and therefore ensures that no loop is formed. Furthermore, the `Gallery` process provides a user interface capability that has some unusual properties. Any incoming communication is always fully acted upon within the process and is not transmitted further. Thus for its inputs the `Gallery` acts as a pure server. For any mouse events that it might generate, the Gallery acts as a pure client provided any event channels are communicated by a channel that utilises an overwriting buffer. This requirement is expounded further in the JCSP documentation.

The operation of a `TargetProcess` is specified as follows. After synchronising on the `goBarrier` it calculates its own random alarm time, which then forms part of an alternative that comprises the alarm and channel communications on its `mousePoints` channel. This alternative is looped around until either the alarm time occurs or the target is hit. In either case the target is no longer active. Another alternative is then entered that comprises communications on its `mousePoints` channel or the `timeAndHitBarrier`. Even though a target is inactive other targets may still not yet have timed out and thus mouse clicks will still be received. The `timeAndHitBarrier` determines when either all the targets have been hit or they have all timed out or some combination of these situations has occurred. It also has the effect of breaking the connection between `TargetController` and `MouseBufferPrompt` until the next set of targets are

initialised. To ensure this does not cause a problem the channel `pointsToTC` uses an `OverWriteOldestBuffer` data store.

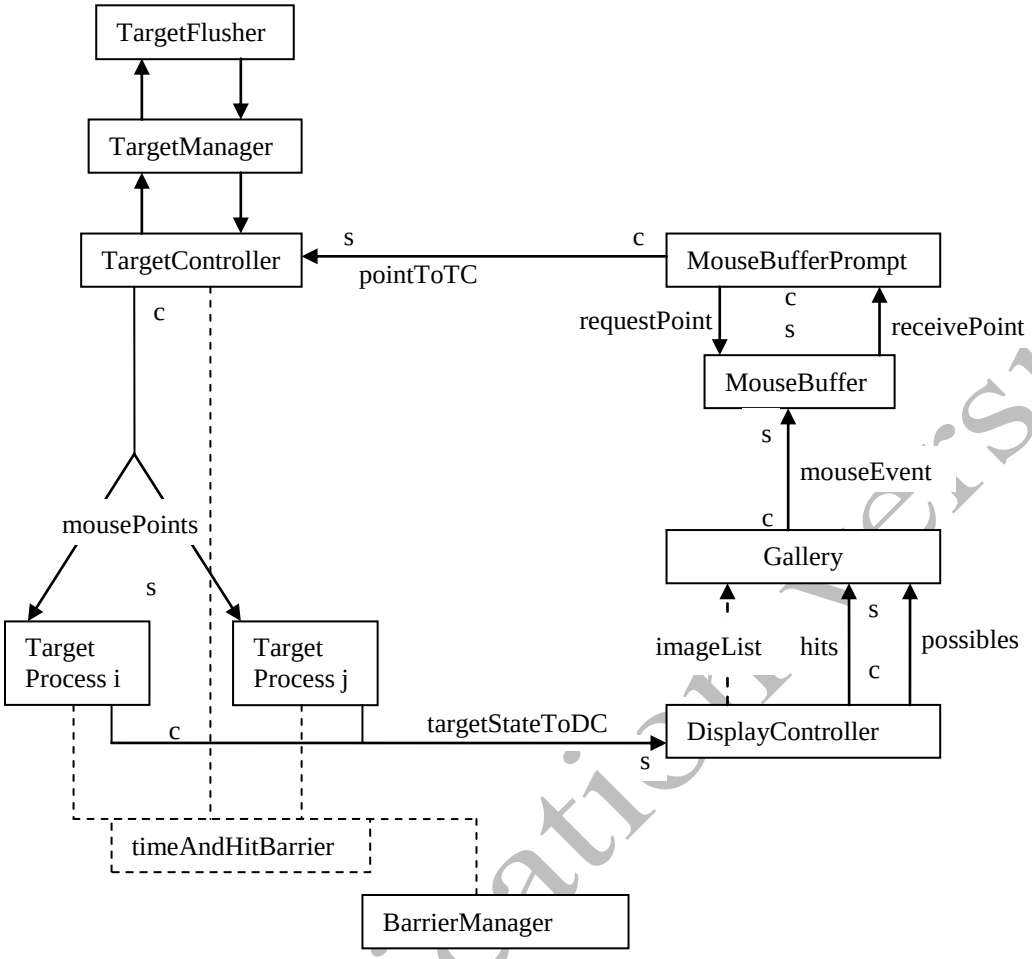

Figure 14-5 System Running Awaiting timeAndHitBarrier

When the state of a target changes (timed out or hit) it sends a communication to the `DisplayController` accordingly, which can then update the display maintained by `Gallery` appropriately. `TargetController` receives a `java.awt.Point` from `MouseBufferPrompt` that give the coordinates where the mouse has been clicked. The `TargetController` then outputs this `Point` value to each of the `TargetProcess` in parallel using the `ChannelOutputList mousePoints`. Once all the targets have either been hit or timed out the `timeAndHitBarrier` synchronises at which point the `TargetProcesses` individually determine into which randomly chosen `Bucket` they are going to fall.

The system then moves on to the final phase of processing shown in Figure 14-6. The DisplayController process contains an alternative with guards comprising the `finalBarrier` and the channel `targetStateToDC`. Thus when it is offering the guard `finalBarrier` together with `BarrierManager` the barrier synchronises and the system is able to progress onto another initial phase as described previously. The only process to undertake any substantial processing in the final phase is the `DisplayController` which leaves the final state of the display for a preset constant time, then sets all the targets to black, thereby obliterating them and the waits for another preset constant time. The coding of each of the processes now follows.

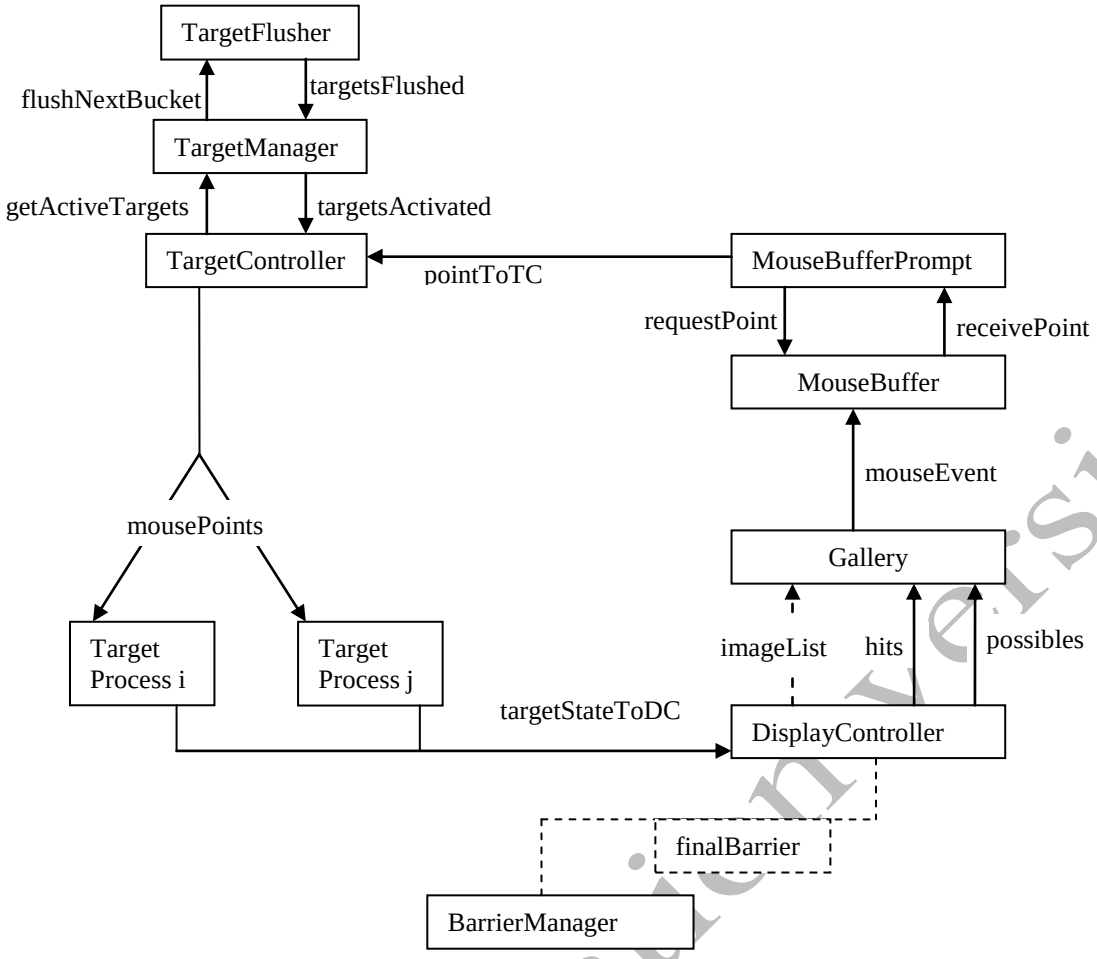

Figure 14-6 System At Final Barrier Synchronisation

14.1 Barrier Manager

The `BarrierManager`, shown in Listing 14-1, simply defines as properties all the barriers in which it participates {2-5}. By definition an `AltingBarrier` must be part of an alternative and thus two `ALT`s are defined {7, 8} in which the particular `AltingBarrier` is the only guard. `BarrierManager` then waits to synchronise on `setUpBarrier` {9}. Thereafter, the process repeatedly synchronises on the `goBarrier`, `timeAndHitBarrier` and `finalBarrier` in sequence {11-13}. A `Barrier` synchronises using the `sync()` method call, whereas synchronisation on an `AltingBarrier` is achieved by calling the `select()` method call of the `ALT` that contains the barrier as a guard. In this case because the guard is the only element in the alternative a simple call of the `select()` method is sufficient, the value returned is of no importance.

```
01     class BarrierManager implements CSProcess {

02       def AltingBarrier timeAndHitBarrier
03       def AltingBarrier finalBarrier
04       def Barrier goBarrier
05       def Barrier setUpBarrier

06       void run() {
07         def timeHitAlt = new ALT ([timeAndHitBarrier])
08         def finalAlt = new ALT ([finalBarrier])
09         setUpBarrier.sync()

10         while (true){
11           goBarrier.sync()
12           def t = timeHitAlt.select()
13           def f = finalAlt.select()
14         }
15       }
16     }
```

Listing 14-1 Barrier Manager

14.2 Target Controller

Listing 14-2 shows the coding of the `TargetController` process, which is the process that effectively controls the operation of the complete system. The properties of the process are defined {18-24} and these directly implement the channel and barrier structures shown in Figures 14-2 to 14-6.

Within the `run` method some constants used to identify guards are defined {27, 28} of an alternative {29}. The zero'th guard of the alternative `controllerAlt` is the `AltingBarrier timeAndHitBarrier` and as such is incorporated into an `ALT` like any other guard. The process then waits for all the other enrolled processes to synchronise on `setUpBarrier` {30} before continuing with the unending loop {31-52} that is the main body of the process.

The first action of the process is to send a signal {32} to the `TargetManager` process using the channel `getActiveTargets`. This is the first part of a client-server request and response pair of communications, the second of which is the receipt of a list of the `targetId`s of the `activeTargets` from the channel `activatedTargets` {33}. The `activeTargets` list is then used to create {36-38} a subset of the `ChannelOutputList` property `sendPoint` {21} in another `ChannelOutputList sendList`, which is used subsequently to communicate with each of the `TargetProcesses`. The Boolean property `active` is then defined {39} and will be used to control the subsequent operation of the process. The process now waits to synchronise on the `goBarrier` {40}. Prior to the `goBarrier` synchronisation all the `TargetProcesses` will have synchronised on the `initBarrier` but that is of no concern to the `TargetController` process.

The `goBarrier` is used to synchronise the operation of all the targets in the running `TargetProcesses`, the `BarrierManager` and the `DisplayController` as well as `TargetController`. The synchronisation enables each of these processes to run in that part of the system which allows users to move their mouse over the active targets and to try and hit each of them, by means of a mouse click, before each target times out. Thus the only actions that can occur are either, a mouse click occurs, or all the targets have either been hit or timed out. The mouse click manifests itself as the input of a `Point` on the `receivePoint` channel {46}. The value of `point` is then communicated, in parallel {47}, to all the members of `sendList` to each of the running `TargetProcesses`. (A `write` on a `ChannelOutputList` causes the writing of the method call parameter to all the channels in the list in parallel). If the barrier guard is `selected` then the loop terminates as soon as all the other processes on the `timeAndHitBarrier` have been selected {43}.

```
17    class TargetController implements CSProcess {

18      def ChannelOutput getActiveTargets
19      def ChannelInput activatedTargets
20      def ChannelInput receivePoint
21      def ChannelOutputList sendPoint
22      def Barrier setUpBarrier
23      def Barrier goBarrier
24      def AltingBarrier timeAndHitBarrier
25      def int targets = 16

26      void run() {
27        def POINT = 1
28        def BARRIER = 0

29        def controllerAlt = new ALT ( [ timeAndHitBarrier, receivePoint] )
30        setUpBarrier.sync()

31        while (true) {
32          getActiveTargets.write(1)
33          def activeTargets = activatedTargets.read()
34          def ChannelOutputList sendList = []
35          for ( t in activeTargets) {
36            sendList.append(sendPoint[t])
37          }
38          def active = true
39          goBarrier.sync()

40          while (active) {
41            switch ( controllerAlt.priSelect() ) {
42              case BARRIER:
43                active = false
44                break
45              case POINT:
46                def point = receivePoint.read()
47                sendList.write(point)
48                break
49            }
50          }
51        }
52      }
53    }
```

Listing 14-2 Target Controller

14.3 Target Manager

Listing 14-3 shows the coding of the `TargetManager` process. Its properties are defined {55-61}. The process does not have anything to do prior to the `setUpBarrier` synchronisation {63}. Its body comprises a non-terminating loop {64-75}. Initially, it reads the signal from `TargetController` on its `getActiveTargets` channel {66}, which causes the writing of yet a further signal to the `TargetFlusher` process on the `flushNextBucket` channel {67}. This is also the first part of the client-server communication pattern between `TargetManager` and `TargetFlusher`. The corresponding response is `read` from the `targetsFlushed` channel, which is the number of `TargetProcesses` that have been initialised into the variable `targetsRunning` {68}. The next phase {69-72} is to read from each of the initialised `TargetProcesses` their identity on the `targetIdFromTarget` channel and append it to the `targetList` {70}. This list is then written to the `TargetController` process {73} using the `activatedTargets` channel, thereby completing the client-server interaction between `TargetManager` and `TargetController`. Finally, the list of initialised targets is written to the `DisplayController` using the channel `activatedTargetsToDC` {74}. These two communications allow the receiving process to complete their initialisation prior to further operation.

```
54     class TargetManager implements CSProcess {

55       def ChannelInput targetIdFromTarget
56       def ChannelInput getActiveTargets
57       def ChannelOutput activatedTargets
58       def ChannelOutput activatedTargetsToDC
59       def ChannelInput targetsFlushed
60       def ChannelOutput flushNextBucket
61       def Barrier setUpBarrier

62        void run() {
63         setUpBarrier.sync()

64         while (true) {
65           def targetList = [ ]
66           getActiveTargets.read()
67           flushNextBucket.write(1)
68           def targetsRunning = targetsFlushed.read()
69           while (targetsRunning > 0) {
70              targetList << targetIdFromTarget.read()
71              targetsRunning = targetsRunning - 1
72           }
73           activatedTargets.write(targetList)
74           activatedTargetsToDC.write(targetList)
75         }
76       }
77     }
```

Listing 14-3 Target Manager

14.4 Target Flusher

The role of the `TargetFlusher` process, shown in Listing 14-4, is to manage the `Buckets` into which the `TargetProcesses` fall. The process also completes the client-server interaction with the `TargetManager` process. Its properties are defined {79-82}. Some variables are initialised {84-86} in the first part of the `run` method. The main loop of the process {87-98} initially reads the signal {88} that causes it to start the initialisation of some `TargetProcesses`. The number of `TargetProcesses` in the `currentBucket` is determined by means of a call of the `holding()` method {89}. The next piece of coding {90-93} ensures that the number of `TargetProcesses` that are `flushed` is greater then zero.

At this stage `initBarrier` can be set to the number of `targetsInBucket` {94} by means of a call to the `reset` method. The number of `targetsInBucket` can now be written to the `TargetManager` process {95}. Now the `TargetProcesses` contained in the `currentBucket` can be `flushed` {96} and therefore start running. Finally, the value of `currentBucket` can be incremented subject to its value staying within zero to the number of Buckets, `nBuckets` {97}.

```
78    class TargetFlusher implements CSProcess {

79      def buckets
80      def ChannelOutput targetsFlushed
81      def ChannelInput flushNextBucket
82      def Barrier initBarrier

83      void run() {
84        def nBuckets = buckets.size()
85        def currentBucket = 0
86        def targetsInBucket = 0

87        while (true) {
88          flushNextBucket.read()
89          targetsInBucket = buckets[currentBucket].holding()
90          while ( targetsInBucket == 0) {
91            currentBucket = (currentBucket + 1) % nBuckets
92            targetsInBucket = buckets[currentBucket].holding()
93          }
94          initBarrier.reset( targetsInBucket)

95          targetsFlushed.write(targetsInBucket)
96          buckets[currentBucket].flush()
97          currentBucket = (currentBucket + 1) % nBuckets
98        }
99      }
100   }
```

Listing 14-4 Target Flusher

14.5 Display Controller

The `DisplayController` process is shown in Listings 14-5 to 14-8 and manages the interaction between the `TargetProcesses` and the user interface provided by the `Gallery` process, described in the next section.

The TargetProcesses communicate with the `DisplayController` by means of the channel `stateChange` {102}, which is the 'One' end of an `Any2One` channel. The channel `activeTargets` {103} is used to input the list of running targets during the initial phase of a cycle. The `displayList` property {104} provides the connection between this process and the `ActiveCanvas` contained in the `Gallery` process. The channels `hitsToGallery` and `possiblesToGallery` {105, 106} are used to send values to the `ActiveLabels` in the `Gallery` process that display the number of targets that have been hit and the total number of targets displayed. Finally, the barriers upon which `DisplayController` synchronises are defined {107-109}.

```
101   class DisplayController implements CSProcess {
102     def ChannelInput stateChange
103     def ChannelInput activeTargets

104     def DisplayList displayList
105     def ChannelOutput hitsToGallery
106     def ChannelOutput possiblesToGallery

107     def Barrier setUpBarrier
108     def Barrier goBarrier
109     def AltingBarrier finalBarrier
```

Listing 14-5 Display Controller Properties

Listing 14-6 gives the array of `GraphicsCommands` and list of values used to change the `displayList`. These are not shown complete, but are those parts that relate to the first and last three targets identified as 0, 1 and 2 and 13, 14 and 15. The array `targetGraphics` is used to initially create the `displayList`. Each of the elements of the list `targetColour` comprises the colour of the target and the element of `targetGraphics` that has to be changed in order to display the target. The first two elements of targetGraphics {113,114} have the effect of completely 'blacking' out the display canvas prior to its repainting within the `Canvas` thread.

```
110     void run() {
111
112       def GraphicsCommand [] targetGraphics = new GraphicsCommand [ 34 ]
113       targetGraphics[0] = new GraphicsCommand.SetColor (Color.BLACK)
114       targetGraphics[1] = new GraphicsCommand.FillRect (0, 0, 450, 450)
115       targetGraphics[2] = new GraphicsCommand.SetColor (Color.BLACK)
116       targetGraphics[3] = new GraphicsCommand.FillRect (10, 10, 100, 100)
117       targetGraphics[4] = new GraphicsCommand.SetColor (Color.BLACK)
118       targetGraphics[5] = new GraphicsCommand.FillRect (120, 10, 100, 100)
119       targetGraphics[6] = new GraphicsCommand.SetColor (Color.BLACK)
120       targetGraphics[7] = new GraphicsCommand.FillRect (230, 10, 100, 100)
121   …
122       targetGraphics[27] = new GraphicsCommand.FillRect (10, 340, 100, 100)
123       targetGraphics[28] = new GraphicsCommand.SetColor (Color.BLACK)
124       targetGraphics[29] = new GraphicsCommand.FillRect (120, 340, 100, 100)
125       targetGraphics[30] = new GraphicsCommand.SetColor (Color.BLACK)
126       targetGraphics[31] = new GraphicsCommand.FillRect (230, 340, 100, 100)
127       targetGraphics[32] = new GraphicsCommand.SetColor (Color.BLACK)
128       targetGraphics[33] = new GraphicsCommand.FillRect (340, 340, 100, 100)
129
130       def targetColour = [
131           [new GraphicsCommand.SetColor (Color.RED), 2],
132           [new GraphicsCommand.SetColor (Color.GREEN), 4],
133           [new GraphicsCommand.SetColor (Color.YELLOW), 6],
134   …
135           [new GraphicsCommand.SetColor (Color.CYAN), 28],
136           [new GraphicsCommand.SetColor (Color.MAGENTA), 30],
137           [new GraphicsCommand.SetColor (Color.ORANGE), 32]   ]
```

Listing 14-6 Graphics definitions

The `run` method has some further properties that are shown in Listing 14-7, which include the constants {138, 139} used to identify the selected alternative defined as `controllerAlt` {142}. The constants {143-145} define the `GraphicsCommand` that can be used to colour a square as indicated by their name. Finally, variables that tally the number of hits and possible hits are defined {146,147} together with a timer {148} that is used to control the time the display stays static at the end of a cycle.

```
138       def BARRIER = 1
139       def CHANGE = 0
140       def TIMED_OUT = 0
141       def HIT = 1
142       def controllerAlt = new ALT ( [stateChange,finalBarrier ] )

143       def whiteSquare = new GraphicsCommand.SetColor(Color.WHITE)
144       def blackSquare = new GraphicsCommand.SetColor(Color.BLACK)
145       def graySquare = new GraphicsCommand.SetColor(Color.GRAY)

146       def totalHits = 0
147       def possibleTargets = 0
148       def timer = new CSTimer()
```

Listing 14-7 Other Run Method Properties

The body of the `run` method is shown in Listing 14-8. Prior to the `setUpBarrier` synchronisation {152} the `displayList` is initialised by a call to the `set` method {149} and the initial, zero, values of `totalHits` and `possibleHits` are written to the `Gallery` {150, 151}.

The never ending loop of the `run` method is then entered {153-187}, which comprises some initialisation prior to the `goBarrier` synchronisation {154-160} followed by the active part of the cycle {162-181} until the `finalBarrier` is selected.

```
149         displayList.set (targetGraphics)
150         hitsToGallery.write (" " + totalHits)
151         possiblesToGallery.write ( " " + possibleTargets )
152         setUpBarrier.sync()

153         while (true) {
154           def active = true
155           def runningTargets = activeTargets.read()  // a list of target Ids
156           possibleTargets = possibleTargets + runningTargets.size
157           possiblesToGallery.write ( " " + possibleTargets )
158           for ( t in runningTargets) {
159             displayList.change ( targetColour[t][0], targetColour[t][1])
160           }
161           goBarrier.sync()

162           while (active) {
163             switch (controllerAlt.priSelect()) {
164               case CHANGE:
165                 def alter = stateChange.read()  // [ tId, state ]
166                 switch (alter [1]) {
167                   case HIT:
168                     displayList.change(whiteSquare, targetColour[alter [0]][1])
169                     totalHits = totalHits + 1
170                     hitsToGallery.write (" " + totalHits)
171                    break
172                   case TIMED_OUT:
173                     displayList.change(graySquare, targetColour[alter [0]][1])
174                     break
175                 }
176                 break
177               case BARRIER:
178                 active = false
179                 break
180             }
181           }
182           timer.sleep(1500)
183           for ( tId in runningTargets ) {
184             displayList.change ( blackSquare, targetColour[tId][1])
185           }
186           timer.sleep ( 500)
187         }
188       }
189     }
```

Listing 14-8 Run Method Definition

The process `DisplayController` is initialised by reading the identities of the running targets into the list `runningTargets` from `TargetManager` using the channel `activeTargets` {155}. The `size` of this list is then used to update the total number of possible targets in the `Gallery` {156-157}. The members of the list are then used to `change` the `displayList`, which cause the targets to appear in the `Gallery` {158-160}. The process then synchronises on the `goBarrier` {161}.

The process remains `active` {162} until the `finalBarrier` is selected {177-179}. It should be noted that the order of the guards in `controllerAlt` is important, with priority given to inputs from the `TargetProcesses`, so that all changes to the targets are completed before the `finalBarrier` is `selected`. While the process is `active`, communications from the running `TargetProcesses` are `read` from the channel `stateChange` {165} which are used to modify the state of the targets in the `Gallery` by `changing` the `displayList`. The input from a `TargetProcess` is in the form of a list comprising the identity of the target and the state to which it should be changed. Two state changes are possible indicated by `HIT`, when the target's image is changed to white {168} and the number of targets hit is also updated {169-170} and `TIMED_OUT` when the square is coloured grey {173}.

Once the `finalBarrier` has been `selected` {163, 178} the process `sleeps` for 1.5 seconds {182} to allow the user to determine the final state of that cycle. The running targets, which are now all coloured either white or grey are returned to the colour black {183-185}. The process `sleeps` for a further 0.5 seconds {186}. It then resumes the main loop of the process, which is initiated by reading the identities of the targets that have been flushed from the next Bucket.

14.6 Gallery

The `Gallery` process shown in Listing 14-9 is similar to other user interface processes that have been discussed previously. The only aspect of particular note is that a mouse event channel is added to the `ActiveCanvas` {217}. There is no need for the programmer to add anything further in terms of listener of event handling methods. Any mouse event is communicated on the `mouseEvent` channel to the `MouseBuffer` process.

```
190   class Gallery implements CSProcess{
191     def ActiveCanvas targetCanvas
192     def ChannelInput hitsFromGallery
193     def ChannelInput possiblesFromGallery
194     def ChannelOutput mouseEvent
195     def canvasSize = 450

196     void run() {
197       def root = new ActiveClosingFrame ("Hand-Eye Co-ordination Test")
198       def mainFrame = root.getActiveFrame()
199       def m1 = new Label ("You Have Hit")
200       def m2 = new Label ("Out Of")
201       def hitLabel = new ActiveLabel (hitsFromGallery)
202       def possLabel = new ActiveLabel (possiblesFromGallery)
203       m1.setAlignment( Label.CENTER)
204       m2.setAlignment( Label.CENTER)
205       hitLabel.setAlignment( Label.CENTER)
206       possLabel.setAlignment( Label.CENTER)
207       m1.setFont(new Font("sans-serif", Font.BOLD, 14))
208       m2.setFont(new Font("sans-serif", Font.BOLD, 14))
209       hitLabel.setFont(new Font("sans-serif", Font.BOLD, 20))
210       possLabel.setFont(new Font("sans-serif", Font.BOLD, 20))
211       def message = new Container()
212       message.setLayout ( new GridLayout ( 1, 4 ) )
213       message.add (m1)
214       message.add (hitLabel)
215       message.add (m2)
216       message.add (possLabel)
217       targetCanvas.addMouseEventChannel ( mouseEvent )
218       mainFrame.setLayout( new BorderLayout() )
219       targetCanvas.setSize (canvasSize, canvasSize)
220       mainFrame.add (targetCanvas, BorderLayout.CENTER)
221       mainFrame.add (message, BorderLayout.SOUTH)
222       mainFrame.pack()
223       mainFrame.setVisible ( true )
224       def network = [ root, targetCanvas, hitLabel, possLabel ]
225       new PAR (network).run()
226     }
227   }
```

Listing 14-9 Gallery Process

14.7 Mouse Buffer

The `MouseBuffer`, shown in Listing 14-10 process `reads` mouse events on its `mouseEvent` channel {248}. Only when the event is a `MOUSE_PRESSED` event does it store the location of the click {251} in the variable `point`. At this stage it modifies {250} the pre-condition on the process' alternative, `mouseBufferAlt` so as to be able to accept requests for a `point` {243}, which can then be transferred to the `MouseBufferPrompt` process {244}, after which the pre-condition is again modified {245} so as not to accept further prompt requests until another mouse click `point` has been received. This mechanism was used previously in the `Queue` and `Event Handling System` and is an idiom or pattern used to manage requests for external non-deterministic events. In this case we note that the `mouseEvent` channel is always available to read events and thus does not block the `Gallery` process with its implicit threads that are used to implement events and a canvas. This is further demonstrated by the `mouseEvent` channel having a data store associated with it that enables the overwriting of the oldest member of the associated buffer (see 16.10).

```
228   class MouseBuffer implements CSProcess{
229     def ChannelInput mouseEvent
230     def ChannelInput getClick
231     def ChannelOutput sendPoint

232     void run() {
233       def mouseBufferAlt = new ALT ( [ getClick, mouseEvent ] )
234       def preCon = new boolean [2]
235       def GET = 0
236       def EVENT = 1
237       preCon[EVENT]= true
238       preCon[GET] = false
239       def point

240       while (true) {
241         switch (mouseBufferAlt. select (preCon)) {
242           case GET:
243             getClick.read()
244             sendPoint.write(point)
245             preCon[GET] = false
246             break
247           case EVENT:
248             def mEvent = mouseEvent.read()
249             if ( mEvent.getID() == MouseEvent.MOUSE_PRESSED) {
250               preCon[GET] = true
251               point = mEvent.getPoint()
252             }
253             break
254         }
255       }
256     }
257   }
```

Listing 14-10 Mouse Buffer Process

14.8 Mouse Buffer Prompt

The `MouseBufferPrompt` process shown in Listing 14-11, simply writes a request to the `getPoint` channel {266} and then waits to read a `point` on the `receivePoint` channel {267} which it then `writes` to the `TargetController` process on the `returnPoint` channel {268}. The combination of `MouseBufferPrompt` and `MouseBuffer` ensures that the `MouseBuffer` process is a pure server in a client-server analysis and also has the effect of decoupling the generation of mouse events in the `Gallery` from the process in which they are consumed, `TargetController`. Furthermore, any delay in reading a `point` by the `TargetController` does not cause a delay that might cause the blocking of the implicit event handling thread of `Gallery`.

```
258   class MouseBufferPrompt implements CSProcess{
259     def ChannelOutput returnPoint
260     def ChannelOutput getPoint
261     def ChannelInput receivePoint
262     def Barrier setUpBarrier

263     void run () {
264       setUpBarrier.sync()

265       while (true) {
266         getPoint.write( 1 )
267         def point = receivePoint.read()
268         returnPoint.write( point )
269       }
270     }
271   }
```

Listing 14-11 Mouse Buffer Prompt Process

14.9 Target Process

The `TargetProcess` is shown in Listings 14-12 to 14-14. The channel `targetRunning` {273} is used by `TargetProcess` to inform the `TargetManager` process that it has been flushed from a `Bucket` and has been made active. The channel `stateToDC` {274} is used to inform the `DisplayController` of any change in state of this target that is, either hit or timed-out. The channel `mousePoint` {275} is used to

input the java.awtPoint at which the mouse has been clicked. The process is a member of the setUp, init, go and timeAndHit barriers {276-279}. It also requires access to the array of buckets {280}. The property targetId {281} is a unique integer identifying the instance of TargetProcess and the values x {282} and y {283} are the pixel co-ordinates of the upper left corner of the target in the display window. The property delay {284} specifies the minimum period for which the target will be displayed before it times out. The target will be visible for a random time between delay and twice delay. The method within {285-293} determines if a java.awt.Point p is within the target area. All targets are square with a side of 100 pixels.

```
272    class TargetProcess implements CSProcess {
273      def ChannelOutput targetRunning
274      def ChannelOutput stateToDC
275      def ChannelInput mousePoint
276      def Barrier setUpBarrier
277      def Barrier initBarrier
278      def Barrier goBarrier
279      def AltingBarrier timeAndHitBarrier
280      def buckets
281      def int targetId
282      def int x
283      def int y
284      def delay = 800

285      def boolean within ( Point p, int x, int y) {
286        def maxX = x + 100
287        def maxY = y + 100
288        if ( p.x < x ) return false
289        if ( p.y < y ) return false
290        if ( p.x > maxX ) return false
291        if ( p.y > maxY ) return false
292        return true
293      }
```

Listing 14-12 The Properties and Within Method of target process

The first part of the run method is executed during the setup phase of the system and is only executed once, Listing 14-13. A Random number generator rng {295} is defined and then used to specify the initial bucket, bucketId {296, 297} into which the TargetProcess will subsequently fall. Initially all TargetProcesses will fall into a bucket in the first half of the array of buckets. A timer and some constants are then defined {298-303}.

Two alternatives are then defined. The alternative preTimeOutAlt {304} is used prior to the TargetProcess being timed out and postTimeOutAlt {305} is used once a time out has occurred or the target has been hit. The latter alternative includes the AltingBarrier timeAndHitBarrier. The operation of such an AltingBarrier is straightforward. It must appear as a guard in an alternative. Whenever any select method on the alternative is called a check is made to determine whether all the other members of the AltingBarrier have also requested and are waiting on such a select. If they have then the AltingBarrier as a whole can be selected. If one of the members of an AltingBarrier accepts another guard in such an alternative then the AltingBarrier cannot be selected. Thus it is possible for a process to offer an AltingBarrier guard and then withdraw from that guard if the dynamics of the system cause that to happen.

The TargetProcess now resigns from timeAndHitBarrier {306}, which at first sight may seem perverse. All TargetProcess are initially enrolled on this barrier. However we only want running targets to be counted as part of the barrier so we must first resign from the barrier and then enroll only when the TargetProcess is executed.

The mechanism of enroll and resign can be applied to all types of barrier. A process that enrolls on a barrier can now call the sync method (Barrier) or be a guard in an alternative and thus can be selected (AltingBarrier). Similarly a process can resign which means that the process is no longer part of the barrier. In the case of a Barrier resignation also implies that if this is the last process to synchronise on the Barrier then this is equivalent to all the processes having synchronised. A process cannot resign if it is not enrolled. In the case of AltingBarriers this enrolment and resignation has to be undertaken with care as no process can be running and selecting the barrier onto which it is intended to either enrol or resign another process from. The associated documentation for JCSP specifies this requirement more fully.

The TargetProcesses now synchronise on the setUpBarrier {307} and when this is achieved they then fallInto the bucket with subscript bucketId {308}. This has the effect of stopping the process. It will only be rescheduled when the TargetFlusher process causes the bucket into which the process has fallen is flushed {96}.

```
294      void run() {
295        def rng = new Random()
296        def int range = buckets.size() / 2
297        def bucketId = rng.nextInt( range )
298        def timer = new CSTimer()
299        def POINT= 1
300        def TIMER = 0
301        def BARRIER = 0
302        def TIMED_OUT = 0
303        def HIT = 1

304        def preTimeOutAlt = new ALT ([ timer, mousePoint ])
305        def postTimeOutAlt = new ALT ([ timeAndHitBarrier, mousePoint ])

306        timeAndHitBarrier.resign()
307        setUpBarrier.sync()
308        buckets[bucketId].fallInto()
```

Listing 14-13 Target process: The Setup Phase of Run

The remainder of the run method, Listing 14-14, only gets executed when the process has been flushed. It comprises a never ending loop {309-349}, which as its final statement {348} causes itself to fall into another bucket, prior to returning to the start of the loop. The loop itself has three phases comprising the phases managed by initBarrier and then that managed by the goBarrier before finally running until either the target is hit or times out which is managed by the timeAndHitBarrier.

In the initial phase, the process enrolls on the timeAndHitBarrier {310} and also the goBarrier {311}. Enrolling on the timeAndHitBarrier causes no problem because at this stage no process can be selecting a guard from an alternative in which timeAndHitBarrier is involved. Similarly, enrolling on the goBarrier is an operation that can be undertaken dynamically because it is a Barrier. The running process now writes its unique identity, targetId to its targetRunning channel {312}. This communication means that the TargetManager now can determine {69-72} which targets are active. It then synchronises on the initBarrier {313}. The number of running TargetProcesses associated with the initBarrier is specified by TargetFlusher {94} at a time when none of these processes can be running because they have yet to be flushed. Only the running TargetProcesses are allowed to access the initBarrier and thus once the initBarrier has synchronised we know that all the TargetProcesses are in the same state and that any dependent processes such as DisplayController will be able to complete any further initialisation prior to the goBarrier synchronisation. The Boolean running is initialised, which will be used subsequently to control the operation of the process. Similarly, the variable resultList is initialised {315} and will be used to indicate the change of state that will occur in the target. The process can now synchronise on the goBarrier by resigning from it {316}. The only permanent members of the goBarrier are BarrierManager, TargetController and DisplayController, all of which simply call the method sync on the barrier {11, 39 and 161}. The goBarrier is augmented by the active TargetProcesses to ensure that all the processes are in a state that will be suitable for the whole system to become active.

Once the process has synchronised on the goBarrier it determines the time for which the target will be displayed and sets the timer alarm {317} which is a guard in the preTimeOutAlt (319}. Prior to the alarm occurring only two things can occur, either the TIMER alarm does happen {320} or a mouse click POINT is received {325}. In the former case, the value TIMED_OUT can be appended to the resultList {322} and this list can be written to the DisplayController using the channel stateToDC {323}. Otherwise, an input can be processed {326} which, if it is within the target area {327} causes the value HIT to be appended to the resultList {329} and as before written to the DisplayController process

{330}. If the point is not `within` the target then the loop repeats until one of the above cases occurs. Once this happens the value of `running` is set `false` {321, 328} and the loop {318-334} terminates.

The process now has take account of the case where other targets are still running; awaiting a time out or a hit, and so mouse clicks and their associated point data will still be received by the `TargetProcess`. Such point data can be ignored {342-344} and only when all the `TargetProcesses` are `selecting` the `timeAndHitBarrier`, together with `TargetController` and `BarrierManager` processes can the `awaiting` loop {335-346} terminate. When this occurs the process `resigns` from the `timeAndHitBarrier` and causes the loop to exit {338-341}.

The TargetProcess can now prepare itself for falling into another bucket by calculating {347} into which bucket it will fall and then calling the `fallInto` method {348}. The chosen bucket is at least two further on than the current bucket which means that it cannot be flushed in the next iteration of `TargetFlusher`, unless the next bucket is empty.

```
309         while (true) {
310           timeAndHitBarrier.enroll()
311           goBarrier.enroll()
312           targetRunning.write(targetId)
313           initBarrier.sync()

314           def running = true
315           def resultList = [targetId]
316           goBarrier.resign()

317           timer.setAlarm( timer.read() + delay + rng.nextInt(delay) )
318           while ( running ) {
319             switch (preTimeOutAlt.priSelect() ) {
320               case TIMER:
321                 running = false
322                 resultList << TIMED_OUT
323                 stateToDC.write(resultList)
324                 break
325               case POINT:
326                 def point = mousePoint.read()
327                 if ( within(point, x, y) ) {
328                   running = false
329                   resultList << HIT
330                   stateToDC.write(resultList)
331                 }
332                 break
333             }
334           }
335           def awaiting = true
336           while (awaiting) {
337             switch (postTimeOutAlt.priSelect() ) {
338               case BARRIER:
339                 awaiting = false
340                 timeAndHitBarrier.resign()
341                 break
342               case POINT:
343                 mousePoint.read()
344                 break
345             }
346           }
347           bucketId = ( bucketId + 2 + rng.nextInt(8) ) % buckets.size()
348           buckets[bucketId].fallInto()
349         }
350       }
351     }
```

Listing 14-14 Target Process: The Active Phase of the Run Method

14.10 Running the System

Listing 14-15 gives the declarations of the channels, barriers and other data required to create the network according to the process network diagrams given in Figures 14-2 to 14-6 and as such are not particularly noteworthy apart from those described below. The `Barriers` are defined with the required number of processes. Thus `setUpBarrier` {357} is defined with the number of `targets` plus five for the other

processes that use this barrier, see Figure 14-2. The `initBarrier` {358} is defined with no members because only the running `TargetProcesses` use this barrier and the number is `reset` explicitly in `TargetFlusher` {94}, see Figure 14-3. Finally, the `goBarrier` {359} is defined has having three members, which are the permanently attached processes as shown in Figure 14-4.

The `AltingBarriers` are defined as an array, with sufficient members such that every process that access them may have a so-called *front-end*. The `finalBarrier` {361} only requires two front-ends because only `BarrierManager` and `DisplayController` participate in this barrier. The barrier `timeAndHitBarrier` {360} requires a front-end for each `TargetProcess`, the `TargetController` and `BarrierManager`. Each process participating in an `AltingBarrier` needs to be allocated its own front-end so that it can access the barrier during an alternative `select` method call. Recall that as a `TargetProcess` becomes active it specifically `enrolls` on the `timeAndHitBarrier` thereby activating its membership of the barrier and when its turn is complete it `resigns` from it. Thus the number of processes that are members of the `timeAndHitBarrier` is determined dynamically at run time. The `Buckets` are defined by means of a `create` method call {362} and this could be any sensible number to provide a wide variety of target initiations per cycle, too many buckets and we would get too few running targets to make the challenge interesting!

The `mouseEvent` channel {363} must be defined with a data store of type `OverWriteOldestBuffer` so that the event handling thread associated with the user interface does not block; see the JCSP documentation for `ActiveCanvas`. Similarly the `pointToTC` channel also uses a one place `OverWriteOldestBuffer` {366} so that if mouse clicks are received too quickly the system does not block. Given the normal performance of a PC this is very unlikely to occur as the user time to move the mouse to another target is relatively long.

The channels that connect `TargetController` to the `TargetProcesses` are defined as any array, `mousePointToTP` {376}, the input end of which is passed directly to the `TargetProcess` {385}. The output ends are formed into a `ChannelOutputList`, `mousePoints` {377}, so that they can be written to in parallel by a `write` method call {47} by `TargetController`.

The `DisplayList` and `ActiveCanvas` components are defined {378-380} prior to being passed as properties of the required processes.

```
352    def targets = 16
353    def targetOrigins = [ [10,  10],[120,  10],[230,  10],[340,  10],
354                          [10, 120],[120, 120],[230, 120],[340, 120],
355                          [10, 230],[120, 230],[230, 230],[340, 230],
356                          [10, 340],[120, 340],[230, 340],[340, 340] ]

357    def setUpBarrier = new Barrier(targets + 5)
358    def initBarrier = new Barrier()
359    def goBarrier= new Barrier(3)

360    AltingBarrier [ ] timeAndHitBarrier = AltingBarrier.create(targets+2)
361    AltingBarrier [ ] finalBarrier = AltingBarrier.create(2)

362    def buckets = Bucket.create(targets)

363    def mouseEvent = Channel.createOne2One ( new OverWriteOldestBuffer(20) )
364    def requestPoint = Channel.createOne2One()
365    def receivePoint = Channel.createOne2One()
366    def pointToTC = Channel.createOne2One( new OverWriteOldestBuffer(1) )

367    def targetsFlushed = Channel.createOne2One()
368    def flushNextBucket = Channel.createOne2One()

369    def targetsActivated = Channel.createOne2One()
370    def targetsActivatedToDC = Channel.createOne2One()
371    def getActiveTargets = Channel.createOne2One()

372    def hitsToGallery = Channel.createOne2One()
373    def possiblesToGallery = Channel.createOne2One()

374    def targetIdToManager = Channel.createAny2One()
375    def targetStateToDC = Channel.createAny2One()

376    One2OneChannel[] mousePointToTP = Channel.createOne2One(targets)
377    def mousePoints = new ChannelOutputList ( mousePointToTP )

378    def imageList = new DisplayList()
379    def targetCanvas = new ActiveCanvas ()
380    targetCanvas.setPaintable ( imageList )
```

Listing 14-15 Running the System Property Definitions

Listing 14-16 shows the definition of the `TargetProcesses` and also of `BarrierManager`. The other processes can be found on the accompanying software because they are very similar to the definition of processes in other systems. It is a matter of tying together the property definition in the process and the defined variable in the script that causes the system to execute. The barriers are straightforward but the allocation of a `timeAndHitBarrier` requires that a specific front-end is allocated to each `TargetProcess` {389} and also to `BarrierManager` {398}. The origin co-ordinates of each `TargetProcess` {392, 393} for the associated display is obtained from the list `targetOrigins` {353-356}.

```
381    def targetList = ( 0 ..< targets ).collect { i ->
382                      return new TargetProcess (
383                          targetRunning: targetIdToManager.out(),
384                          stateToDC: targetStateToDC.out(),
385                          mousePoint: mousePointToTP[i].in(),
386                          setUpBarrier: setUpBarrier,
387                          initBarrier: initBarrier,
388                          goBarrier: goBarrier,
389                          timeAndHitBarrier: timeAndHitBarrier[i],
390                          buckets: buckets,
391                          targetId: i,
392                          x: targetOrigins[i][0],
393                          y: targetOrigins[i][1],
394                          delay: 2500
395                          )
396                      }

397    def barrierManager = new BarrierManager (
398                          timeAndHitBarrier: timeAndHitBarrier[targets],
399                          finalBarrier: finalBarrier[0] ,
400                          goBarrier: goBarrier,
401                          setUpBarrier: setUpBarrier
402                          )
```

Listing 14-16 Decalring the TargetProcesses and BarrierManager

14.11 Summary

This chapter has introduced the concepts of buckets and barriers as a means of providing synchronisation between processes that are executing on a single processor within a single JVM. It has been shown how an `AltingBarrier` can be used to manage highly dynamic situations and to provide a high-level control mechanism to manage complex interactions. A description of the implementation mechanism underlying `AltingBarrier` is to be found in [Welch CPA 2007] and a different use of `AltingBarrier` using a syntactically different but conceptually identical formulation is to be found in [Ritson CPA 2007].