9 External Events: Handling Data Multiple Sources

Traditionally, real-time systems that respond to external stimuli have utilised interrupts[1]. Over the years a great deal of effort has been expended in trying to make interrupt based systems more efficient and easier to program. However, the basic problem still remains that an interrupt causes the halting of the current program, saving its state and then starting an interrupt service routine. The problems become more complex when an interrupt service routine is itself interrupted by a device with a higher priority. The approach has been to reduce the amount of time when interrupts are disabled. This in itself leads to further problems because it is very difficult then to foresee the precise nature of interactions between interrupts that can then take place. It is these indeterminate interactions that cause problems when systems are running because it is impossible to test for all the possible interactions, especially in highly complex systems.

The framework built so far, uses parallelism and alternation to capture non-deterministic behaviour, provides a means of describing, implementing and analysing such event driven systems. Rather than building a system that interrupts itself on receipt of an event notification; build a system that expects such events to occur so that programmers can better reason about its behaviour. In effect, the external event is considered to be the same as a channel communication. Furthermore, the client-server design pattern gives us a handle by which the system can be analysed to ensure that unwanted interactions between events do not occur.

9.1 An Event Handling Design Pattern

The aim of the pattern is to allow the system to respond to external events as quickly as possible. However, the situation has to be considered that events may occur so rapidly that the system cannot deal with all the events. Such a situation tends to overwhelm interrupt based systems. The pattern also has to take account of any priority requirement the application may have, thereby influencing the order in which events are handled. Such ordering of the handling of events may result in some events being lost. However, if the designer is aware of this situation then steps can be taken at design time to ameliorate their effects.

The key to building an event handling system is that the process dealing with receipt of the event has to be ready, waiting, for the associated channel (event) communication, so it can be read and the associated data passed on to another process. The event receiving process can then return to the state of waiting for the next event communication. If we connect the event receiving process directly to the event processing process then the event receiver might be delayed by having to waiting for the processing of another event to finish. We thus require an intermediate stage that separates event receiving from event processing.

[1] Interrupt: a hardware signal that indicates that a device needs to be serviced and which causes the processors' central processing unit to interrupt the current program and invoke the device's service routine returning to the original program once the device has been serviced.

Pre-publication version

This can be implemented as some form of buffer. More specifically, the buffer should always be ready to receive a communication from the event receiver process. This may mean that previous buffered values may be overwritten. In addition, a mechanism by which buffered values can be requested from the buffer process, in a manner similar to that used in the Queue process described in Chapter 6. The resulting process structure is shown in Figure 9-1, to which a client-server labelling has been added. It demonstrates that there are no client-server loops in the architecture.

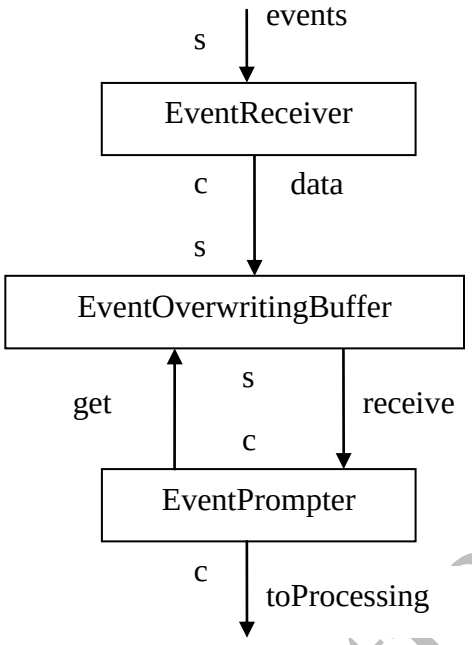

Figure 9-1 Event Handling Design Pattern

Events are received by the EventReceiver and immediately sent to the EventOverwritingBuffer so that EventReceiver is ready to read the next event. The EventPrompter indicates that it wants to get some data, which it will receive immediately from the EventOverwritingBuffer if data is already buffered or it will have to wait until an event has been input. The EventPrompter then writes the data to the rest of the system where it is processed. Thus it is EventPrompter that has to wait until subsequent processing can be undertaken, allowing EventReceiver always ready to read an event. Later we shall show that the time required to process events can in fact be calculated to give an absolute upper bound on the performance of the system. Such a bound cannot be calculated for interrupt based systems. In addition, the client-server labelling shows that the pattern has no deadlock or livelock inherent within it and thus provided the rest of the system is also deadlock and livelock free ensures that the system will behave as expected. This is easily deduced because the EventOverwritingBuffer is a pure server and hence any client-server circuit cannot exist.

9.2 Utilising the Event Handing Pattern

The pattern can be transformed easily into a set of processes that achieves its effect.

9.2.1 The EventReceiver Process

Listing 9-1 shows the definition of the `EventReceiver` process. The process has `eventIn` {2} and `eventOut` {3} channel properties. In this implementation, every input that is `read` from `eventIn` is immediately `written` to the `eventOut` channel {6}. In a realistic implementation it would probably be necessary to determine the source of the event and possibly read some data from a hardware register. However, we can presume that such additional processing would not create any delay within the system because the event would not be raised if there was no reason.

Pre-publication version

```
01     class EventReceiver implements CSProcess {

02       def ChannelInput eventIn
03       def ChannelOutput eventOut

04       void run() {
05         while (true){
06           eventOut.write(eventIn.read())
07         }
08       }
09     }
```

Listing 9-1 The EventReceiver Process

9.2.2 The Event Overwriting Buffer Process

The implementation of EventOWBuffer is shown in Listing 9-2.

```
10     class EventOWBuffer implements CSProcess {

11       def ChannelInput inChannel
12       def ChannelInput getChannel
13       def ChannelOutput outChannel

14       def void run () {
15         def owbAlt = new ALT ( [inChannel, getChannel] )
16         def INCHANNEL = 0
17         def GETCHANNEL = 1
18         def preCon = new boolean[2]
19         preCon[INCHANNEL] = true
20         preCon[GETCHANNEL] = false
21         def e = new EventData ()
22         def missed = -1

23         while (true) {
24           def index = owbAlt.priSelect ( preCon )
25           switch ( index ) {
26             case INCHANNEL:
27               e = inChannel.read().copy()
28               missed = missed + 1
29               e.missed = missed
30               preCon[GETCHANNEL] = true
31               break
32             case GETCHANNEL:
33               def s = getChannel.read()
34               outChannel.write ( e )
35               missed = -1
36               preCon[GETCHANNEL] = false
37               break
38           }
39         }
40       }
41     }
```

Listing 9-2 The EventOWBuffer Process

The channel `inChannel` {11} inputs data from the `EventReceiver` process. The `getChannel` {12} receives a signal from `EventPrompter` whenever that process requires data. The response to `EventPrompter` is output on the channel `outChannel` {13}. The process receives inputs on its input channels over which it must alternate as the order in which such inputs are read cannot be determined. This is captured in the definition of `owbAlt` {15}. The `EventOWBuffer` also has to capture the behaviour that requests for data from the `EventPrompter` process can only be allowed when the buffer contains data. To this end we used pre-conditions on `owbAlt` in a manner similar to that used in the `Queue` process described in Chapter 6. The constants `INCHANNEL` {16} and `GETCHANNEL` {17} are used to access the elements of the `preCon` {18} Boolean array and also to identify the `case`s within the `switch` that implements the main processing loop. The initial values of the `preCon` elements can be specified as follows. The process is always willing to accept inputs on its `inChannel` and thus this element is always `true` {19}. Initially there is no data in the buffer and thus requests to get data from `EventPrompter` must not be permitted and thus that pre-condition has to be set `false` {20}. The actual buffer is represented by the variable `e` {21} and is of type EventData, see Listing 9-4. The variable `missed` {22} will count the

Pre-publication version

number of times the data in the buffer `e` was overwritten and will be passed through the system so that its performance can be analysed. It is initialised to -1 so that when the next event is read its value will be considered not have been overwritten because the value of `missed` will be 0.

The main loop of EventOWBuffer {23-37} initially determines the `index` of the enabled channel, with priority being given to `inChannel` {24} because we always want `EventReceiver` to be ready to read the next event. In that case, the event data is read from `inChannel` {27} and a deep copy is made into the buffer variable `e`. The interface `JCSPCopy`, defined in `org.jcsp.groovy`, defines an abstract method `copy()` that can be used to make a deep copy of an object. Recall that if an object is transferred from one process to another then if these processes are on the same process then this communication is achieved by passing an object reference. We must ensure that two processes do not access the same object at the same time and hence the need to make a deep copy of the object. The value of `missed` is incremented {28} and saved in the buffer variable `e` {29}. The `preCon` element `GETCHANNEL` can now be set `true` {30} because there is data in the buffer that can be sent to `EventPrompter` following a request for data.

Once the buffer contains data then requests for data can be read from the `getChannel` {33} and the contents of the buffer are immediately written to the `outChannel` {34}. This interaction ensures the process behaves like a server. The `preCon` element `GETCHANNEL` must now be set `false` {36} because there is no longer any data in the buffer and likewise the variable `missed` must be reset to -1 {35}.

9.2.3 The Event Prompter Process

This process is shown in Listing 9-3. This process has channel properties {43-45} that reflect the process structure shown in Figure 9-1. A signal is written to the `getChannel` {48}, the completion of which may be delayed until the `EventOWBuffer` contains event data. The response from EventOWBuffer is immediately read into a variable `e` {49} and also uses the `copy` method to ensure that the data cannot be modified as it resides within the `EventPrompter` before being output to the next process. The data is then written {50} to the `outChannel`, where yet again a delay may be incurred due to the processing system not being in a state where the data from this event source can be processed.

```
42    class EventPrompter implements CSProcess {

43      def ChannelInput inChannel
44      def ChannelOutput getChannel
45      def ChannelOutput outChannel

46      def void run () {
47        while (true) {
48          getChannel.write(1)
49          def e = inChannel.read().copy()
50          outChannel.write( e )
51        }
52      }
53    }
```

Listing 9-3 The EventPrompter Process

9.2.4 The EventData Class

`EventData` contains three properties for this explanatory description comprising `source` {55}, `data` {56} and `missed` {57}. The class implements both the `Serializable` interface so that `EventData` objects can be communicated over networks and `JCSPCopy` so that the `copy` method {58-63} can be defined that makes deep copies of `EventData` objects. A `toString` method has also been provided so that event data can be more easily output {64-70}.

Pre-publication version

```
54    class EventData implements Serializable, JCSPCopy {

55      def int source
56      def int data
57      def int missed

58      def copy() {
59        def e = new EventData ( source: this.source,
60                                data: this.data,
61                                missed: this.missed )
62        return e
63      }

64      def String toString() {
65        def s = "EventData -> [source: "
66        s = s + source + ", data: "
67        s = s + data + ", missed: "
68        s = s + missed + "]"
69        return s
70      }

71    }
```

Listing 9-4 The EventData Class Definition

9.3 Analysing Performance Bounds

The ability of the design pattern to handle repeated events can be determined for two different cases. The first and simplest case occurs when there is no outstanding request for data from the `EventPrompter` process. The time to handle an event can be calculated by adding together the processing times of lines 6, 24, 25 and 27 to 31 plus the time to undertake a single communication from `EventReceiver` to `EventOWBuffer`. This value can be determined by calculation if the time to execute each statement can be determined.

The second case is slightly more complex and concerns the situation when `EventOWBuffer` has just accepted a request to get data from `EventPrompter` and an event arrives at `EventReceiver`. The required processing comprises lines 33 to 37, 49 and 24 plus the time taken to undertake a single communication from `EventOWBuffer` to `EventPrompter`. Line 24 is the selection of the alternative, which is bound to select the event input because it is given the highest priority. Thus this time, plus the time to actually process the event, which is the same as the first case, gives the total time that is required to handle an event. This therefore gives an upper bound for the time to process an event and thus the maximum rate at which events can be handled in the worst case scenario. On a modern processor these times will be measured in microseconds. The fact that the processing system might not be able to keep up with such a rate merely points to a deficiency in the system design and not a failure of the ability to use parallel processing techniques to handle events.

9.4 Demonstration of the Event Handling System

The demonstration comprises an `EventHandler` process that is the parallel composition of the processes previously described and shown in Figure 9-1. This is fed with 'events' by an `EventGenerator` process that outputs data values according to a uniformly distributed delay strategy.

9.4.1 The Event Generator Process

The `EventGenerator` process, shown in Listing 9-5, comprises two processes, `EventStream` and `UniformlyDistributedDelay`. The properties of the process are passed directly to these processes and will thus be described in the next sections.

Pre-publication version

```
72     class EventGenerator implements CSProcess {

73       def ChannelOutput outChannel
74       def int source = 0
75       def int initialValue = 0
76       def int minTime = 100
77       def int maxTime = 1000
78       def int iterations = 10

79       def void run () {

80         One2OneChannel es2udd = Channel.createOne2One()

81         println "Event Generator for source ${source} has started"

82         def eventGeneratorList = [
83                 new EventStream ( source: source,
84                                   initialValue: initialValue,
85                                   iterations: iterations,
86                                   outChannel: es2udd.out() ),
87                 new UniformlyDistributedDelay ( minTime: minTime,
88                                                 maxTime: maxTime,
89                                                 inChannel: es2udd.in(),
90                                                 outChannel: outChannel )
91                 ]
92         new PAR (eventGeneratorList).run()
93       }
94     }
```

Listing 9-5 The EventGenerator Process

9.4.2 The Event Stream Process

Listing 9-6 shows the `EventStream` process, in which the `source` property {96} is used to identify the stream and which has an `initialValue` {97}. The process will output a stream of length `iterations` {98}. The stream of 'events' will be output on the channel `outChannel` {99}. The process uses the `upto` method to create the loop {102}. An event `e` is constructed {103} and then written to `outChannel` {104}. On completion the process outputs a message {107} as this will prove invaluable in understanding how the system functions.

```
95     class EventStream implements CSProcess {

96       def int source = 0
97       def int initialValue = 0
98       def int iterations = 10
99       def ChannelOutput outChannel

100      def void run () {
101        def i = initialValue
102        1.upto(iterations) {
103          def e = new EventData ( source: source, data: i )
104          outChannel.write(e)
105          i = i + 1
106        }
107        println "Source $source has finished"
108      }
109    }
```

Listing 9-6 The EventStream Process

9.4.3 The Uniformly Distributed Delay Process

The `UniformlyDistributedDelay` process, shown in Listing 9-7, uses a random number generator {117} to produce a `delay` between `minTime` and `maxTime` {120}. The event data is read from `inChannel` {119} and after waiting for the `delay` {121} period it is output on `outChannel` {122}.

Pre-publication version

```
110   class UniformlyDistributedDelay implements CSProcess {

111     def ChannelInput inChannel
112     def ChannelOutput outChannel
113     def int minTime = 100
114     def int maxTime = 1000

115     def void run () {
116       def timer = new CSTimer()
117       def rng = new Random()
118       while (true) {
119         def v = inChannel.read().copy()
120         def delay = minTime + rng.nextInt ( maxTime - minTime )
121         timer.sleep (delay)
122         outChannel.write( v )
123       }
124     }
125   }
```

Listing 9-7 The UniformlyDistributedDelay Process

9.4.4 Demonstration of the Event Processing System

The script that invokes the system is shown in Listing 9-8. The collection of processes comprises the processes already described, executed in parallel. An additional `UniformlyDistributedDelay` process has been included to represent the varying time it takes to process an event. The events are passed to a `GPrint` process where they are simply printed. Of particular interest is the number of events that are missed.

The events are generated with a delay that varies between 100 and 200 milliseconds {136, 137}. The simulation of processing time {142, 143} between 1000 and 2000 milliseconds means we would expect around 8 or 9 events to be missed but will depend on the actual random. A sample output from the system is shown in Output 9-1.

The first two events pass through the system without any delay because that is the time when the buffers within the system are being filled. Thereafter, data appears with varying numbers of events missed and in general these match what would be expected. The last three events are produced after the event generator has finished because they are buffered up within the system. It should be noted that a check of correctness of operation is possible because the data value, after the first, is equal to the previous output data value plus the number missed plus 1.

```
126   One2OneChannel eg2h = Channel.createOne2One()
127   One2OneChannel h2udd = Channel.createOne2One()
128   One2OneChannel udd2prn = Channel.createOne2One()
129
130
131   def eventTestList = [
132         new EventGenerator ( source: 1,
133                              initialValue: 100,
134                              iterations: 100,
135                              outChannel: eg2h.out(),
136                              minTime: 100,
137                              maxTime:200 ),
138         new EventHandler ( inChannel: eg2h.in(),
139                            outChannel: h2udd.out() ),
140         new UniformlyDistributedDelay ( inChannel:h2udd.in(),
141                                         outChannel: udd2prn.out(),
142                                         minTime: 1000,
143                                         maxTime: 2000 ),
144         new GPrint ( inChannel: udd2prn.in(),
145                      heading : "Event Output",
146                      delay: 0)
147         ]
148   new PAR ( eventTestList ).run()
```

Listing 9-8 The Srcipt Used to Invoke the Event Handling System

Pre-publication version

```
Event Output
Event Generator for source 1 has started
EventData -> [source: 1, data: 100, missed: 0]
EventData -> [source: 1, data: 101, missed: 0]
EventData -> [source: 1, data: 110, missed: 8]
EventData -> [source: 1, data: 122, missed: 11]
EventData -> [source: 1, data: 128, missed: 5]
EventData -> [source: 1, data: 140, missed: 11]
EventData -> [source: 1, data: 149, missed: 8]
EventData -> [source: 1, data: 159, missed: 9]
EventData -> [source: 1, data: 168, missed: 8]
EventData -> [source: 1, data: 176, missed: 7]
Source 1 has finished
EventData -> [source: 1, data: 186, missed: 9]
EventData -> [source: 1, data: 195, missed: 8]
EventData -> [source: 1, data: 199, missed: 3]
```

Output 9-1 A Sample Output from the Event Handling System

9.5 Summary

This chapter has shown that the adoption of parallel processing design techniques and implementation can shed a new light on age old computing problems. In particular, it allows designers to reason about both a system's behaviour and its performance when subjected to a large number of randomly occurring events.

9.6 Exercises

1. Using the suggestion made earlier in the chapter construct a `GroovyTestCase` for the event handling system.

1. The accompanying web site contains a version of the event handling system, `MultiStreamTest`, which allows the creation of 1 to 9 event streams. By modifying the times associated with each event generation stream and also of the processing system explore the performance of the system. What do you conclude?

2. The process `EventProcessing` has three versions of multiplexer defined within it, two of which are commented out. By choosing each of the options in turn, comment upon the effect that each multiplexer variation has on overall system performance.

3. A manufacturing process utilises hoppers and a blender. The hoppers are used hold raw materials and the blender is used to mix the contents from one or more hoppers. The collection of hoppers and the blender is managed by a controller. The hoppers indicate when they are ready to be used. The blender indicates when it is ready and also when mixing is to stop. The hoppers and the blender are clients to the server manager of the controller. The hoppers nake a request to the manager to determine when they should stop processing raw materials. The aim of this exercise is to create three different control regimes as follows:

   i) The hoppers and the blender indicate they are ready to start but mixing only commences when all three hoppers are ready after which the ready signal from the blender is ready.

   ii) As in (i) above but mixing commences as soon as two hoppers and the blender are ready. If three hoppers are ready before the blender is ready then the last hopper is not used and will only be used during the following mixing cycle.

   iii) As in (ii) above but mixing commences as soon as just one hopper and the blender are ready. If more than one hopper is ready before the blender becomes ready then these are retained untilthe following mixing cycle(s).

Pre-publication version

The accompanying web site has definitions for Hopper and Blender processes that utilise the GConsole to enable user interaction. These processes are complete and implement a client style behaviour. The user inputs an 'r' into the input area of the GConsole of the required Hopper or Blender to signify that it is ready. The Hopper or Blender process then outputs a'1' to signal to the Manager process that it is ready. The Manager then implements the required control regime as described above. A Hopper process then sends a '2' signal to the Manager indicating that it is ready to be stopped. The Blender process waits for the userto input an 'f' to indicate that blending can stop. The Blender then sends a'2' to the Manager process. The Manager then completes the client-server interactions. The web site contains scripts to execute each of the above control regimes. There are also outline process definitions for each of the control regimes that need to be completed. Initially, you are advised to produce a process network diagram to enablea betterunderstanding of the interactions and process architecture.

Pre-publication version