A Code Generation Example for Event-B: A Shared Channel with Concurrent Read/Writers

Andy Edmunds and Michael Butler
University of Southampton

July, 2009

1 Introduction

The example that we present here involves processes reading and writing to a shared channel. A channel may have at most one reader reading, and at most one writer writing at any one time; however a number of processes may be waiting to read from, or write to, the channel. In our most abstract model data is transferred as a block in a single atomic step. A write event constitutes moving a block from a writing process to a channel buffer; and a read event constitutes moving a block from a channel buffer to a reader. The atomicity of the read and write activity is altered in the refinement - we introduce blocks that are made up of packets, and each packet is written to the channel individually. This allows the reader to begin reading as soon as there is data in the channel - without the writer having to complete the data transfer. We continue with an overview of our implementation notation (OCB), and show the implementation level specification for the concurrent read/writers example. We then present details of the implementation level refinement that results from the translation of the OCB model into Event-B, and also show details of the source code generated by the Java translator. More details of the approach can be found in our paper [2].

2 The Abstract Event-B Development

In our first model we introduce processes, channels and data. We define carrier sets for the set of processes $Process$, the set of channels $Channel$, and set of data blocks represented by $Block$. A block of data is a function of packet identifiers to data, $Block = PKTID \pfun DATA$. Process objects are represented by a variable *proc*, and channels are represented by a variable *chan*. Each process

has a local buffer called $buff$, and channels hold data in a buffer called $data$.

INVARIANTS
$proc \subseteq PROCESS$
$chan \subseteq CHANNEL$
$data \in chan \rightarrow Block$
$buff \in proc \rightarrow Block$

The $Write$ event models the write of a block of data b to a channel. The parameters model the writing process p, and the target channel c. In addition to these parameters an additional local parameter b is introduced to keep track of the block of data to be written from the local buffer, where $b = buff(p)$.

$Write \triangleq$
ANY $p,\ c,\ b$
WHERE $p \in proc\ \wedge\ c \in chan\ \wedge b = buff(p)\ \wedge$
$buff(p) \neq \varnothing\ \wedge\ data(c) = \varnothing$
THEN $data(c) := b\ \parallel\ buff(p) := \varnothing$
END

The block b is copied to the data buffer $data(c)$ and the local buffer $buff(p)$ is emptied.

The event to read a block from the channel is parameterized by a reading process p, and channel c.

$Read \triangleq$
ANY $p,\ c,\ b$
WHERE $p \in proc\ \wedge\ c \in chan\ \wedge\ b = data(c)$
$buff(p) = \varnothing\ \wedge\ data(c) \neq \varnothing$
THEN $data(c) := \varnothing\ \parallel\ buff(p) := b$
END

The block in the channel buffer b is copied to the local buffer $buff(p)$ and the channel buffer $data(c)$ is emptied.

3 Refinement with Packetized Data

In the first refinement we introduce writing behaviour which is performed in three steps by the $StartWrite$, $WritePacket$, and $EndWrite$ events. Similarly $StartRead$, $ReadPacket$ and $EndRead$ perform reading. We can make use of a

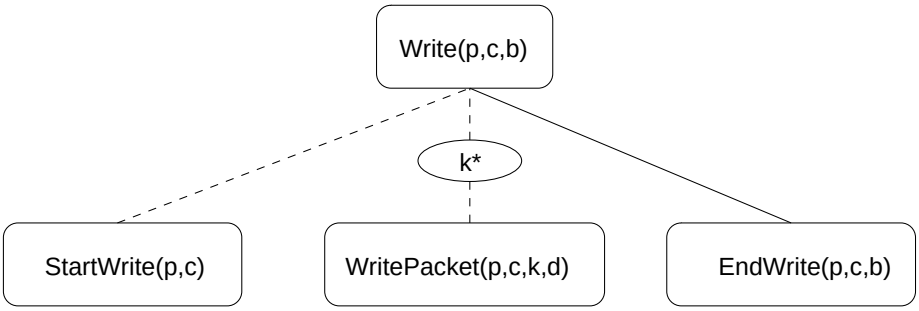

Figure 1: Decomposing the Write Event

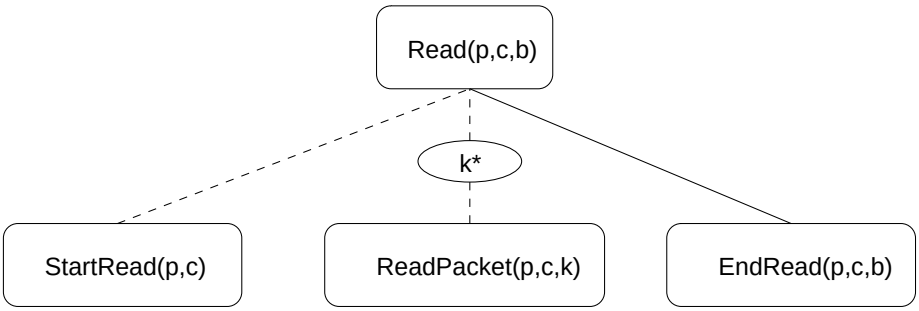

Figure 2: Decomposing the Read Event

graphical representation, introduced in [1], called Event Refinement Diagrams. These diagrams are based on Jackson Structure Diagrams, and are used to clarify the relationships between events of the abstraction and refinement. It should be noted though that the diagrams are an informal representation of the relationship between abstract events and events of refinements. The most abstract specification appears uppermost in the diagram with more concrete representations below. At each level the order of events is read from left to right, indicating the sequence in which the events are required to occur. The solid lines connecting events represent event refinement. Dashed lines represent events that refine skip, and add behaviour related to the abstract event. The '*' annotation indicates that iteration of a particular event is possible. We indicate the parameter names of each event in the form of an event signature. In Figures 1 and 2, k^* indicates that the number of iterations is determined by a guard involving parameter k. The diagrams show abstract *Read* and $Write$ events being refined by a number of events.

We introduce $buff2$, a local buffer, where data can be added or removed one packet at a time; and $data2$ which is a channel buffer (where data can also

be added or removed one packet at a time).

Invariants

$writing \in proc \rightarrowtail chan$

$reading \in proc \rightarrowtail chan$

$buff2 \in proc \rightarrow Block$

$data2 \in chan \rightarrow Block$

We ensure processes cannot be reading and writing at the same time with the invariant,

$$dom(writing) \cap dom(reading) = \varnothing$$

However we allow channels to be in the range of both *reading* and *writing* simultaneously.

The added events: $StartWrite$ refines skip.

$StartWrite \triangleq$
ANY $p,\ c$
WHERE $p \in proc\ \wedge\ c \in chan\ \wedge\ p \notin dom(writing)\ \wedge$
$c \notin ran(writing)\ \wedge\ buff2(p) \neq \varnothing\ \wedge\ data2(c) = \varnothing \wedge$
$p \notin dom(reading)$
THEN $writing := writing \cup \{p \mapsto c\}$
END

The process and channel $p \mapsto c$ are added to the set of writing pairs. Once a process-channel pair are added to the set of writing pairs we transfer individual packets of data $k \mapsto d$, from one to the other. k is the packet identifier and d represents the data.

The $WritePacket$ event:

$WritePacket \triangleq$
ANY $p,\ c,\ k,\ d$
WHERE $p \mapsto c \in writing\ \wedge\ k \in dom(buff2(p))\ \wedge$
$d = buff2(p)(k)\ \wedge\ k \notin dom(data2(c))$
THEN $data2(c) := data2(c) \cup \{k \mapsto d\}$
END

In the $WritePacket$ event a packet $k \mapsto d$ is added to the channel buffer $data2(c)$.

The *EndWrite* event refines $Write$.

$EndWrite \triangleq$
REFINES $Write$
ANY $p,\ c$
WHERE $p \mapsto c \in writing \ \land \ c \in chan \ \land \ data2(c) = buff2(p)$
WITH $b = buff2(p)$
THEN $writing := \{p\} \lhd writing \parallel buff2(p) := \varnothing$
END

The process p is removed from the set of writers $writing := \{p\} \lhd writing$ and the local buffer is cleared, $buff2(p) := \varnothing$.

StartRead refines skip:

$StartRead \triangleq$
ANY $p,\ c$
WHERE $p \in proc \ \land \ c \in chan \ \land \ p \notin dom(reading) \ \land$
$c \notin ran(reading) \ \land \ p \notin dom(writing) \ \land \ data2(c) \neq \varnothing \ \land \ buff2(p) = \varnothing$
THEN $reading \cup \{p \mapsto c\}$
END

The process and channel pair $p \mapsto c$ are added to the set of reading pairs.

ReadPacket refines skip:

$ReadPacket \triangleq$
ANY $p,\ c,\ k$
WHERE $p \mapsto c \in reading \ \land \ k \in dom(data2(c)) \ \land \ k \notin dom(buff2(p))$
THEN $buff2(p) := buff2(p) \cup \{k \mapsto data2(c)(k)\}$
END

A packet from the channel buffer, represented by $k \mapsto data2(c)(k)$, is added to the local buffer $buff2(p)$.

EndRead refines *Read*:

$$
\begin{array}{l}
EndRead \triangleq \\
\quad \textbf{REFINES } Read \\
\quad \textbf{ANY } p,\ c \\
\quad \textbf{WHERE } p \in proc \ \wedge \ c \in chan \ \wedge p \mapsto c \in reading \ \wedge \\
\qquad c \notin ran(writing) \ \wedge \ buff2(p) = data2(c) \\
\quad \textbf{WITH } b = data2(c) \\
\quad \textbf{THEN } data2(c) := \varnothing \ \parallel \ reading := reading \setminus \{p \mapsto c\} \\
\quad \textbf{END}
\end{array}
$$

The channel buffer is emptied in the event action, $data2(c) := \varnothing$, the data is considered to have been consumed by the reading process. The process-channel pair is removed from the set of reading pairs, $reading := reading \setminus \{p \mapsto c\}$.

Gluing invariants added: The channel data block *data* is equal to the packetized data *data*2, except when the process is writing.

$$\forall c \cdot c \in chan \wedge c \notin ran(writing) \Rightarrow data(c) = data2(c)$$

Show that the process block buffer $buff$ must be the same as the packetized buffer $buff2$, except when the process is reading.

$$\forall p \cdot p \in proc \wedge p \notin dom(reading) \Rightarrow buff(p) = buff2(p)$$

4 A Brief Introduction to OCB

Object-oriented Concurrent-B (OCB) is a specification notation used to link Event-B and an object-oriented programming language for implementation. Our system may consist of a number of processes which may perform some tasks, and some objects which may be shared, where mutually exclusive access to data is enforced. A specification consists of process and monitor classes. Process classes allow specification of interleaving behaviour, using non-atomic constructs, where atomic regions are defined by labelled atomic clauses. Monitor classes may be shared between the processes, they provide mutually exclusive access to the shared data using atomic procedures, and may incorporate conditional waiting. We specify sequences of atomic clauses using a semi-colon operator. A simple example of a non-atomic clause, with two labelled atomic clauses which update variables x and y, follows:

$$label1 : x := 0; label2 : y := 0$$

Processes are able to interleave in a non-atomic clause where a semi-colon is specified. Each labelled atomic clause maps to an event guarded by a program counter which is derived from the label. This allows us to model ordered the

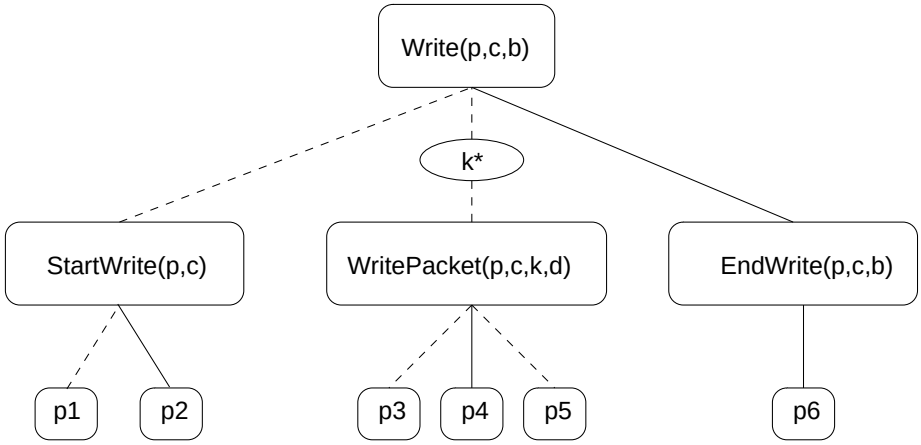

Figure 3: Implementing the Write Event

execution of an implementation. In addition to the semi-colon operator we have a branching construct,

$$\textbf{if}(g)\ \textbf{then}\ a\ [\ \textbf{andthen}\ na\ ]\ \textbf{endif}$$

where g is a guard, a is an assignment action or procedure call, and na is an optional non-atomic clause. In a branching clause g and a form an atomic guarded action. This may optionally be followed by a non-atomic clause or additional branches. Each conditional branch maps to a guarded event. The looping construct follows,

$$\textbf{while}(g)\ \textbf{do}\ a\ [\ \textbf{andthen}\ na\ ]\ \textbf{endwhile}$$

As in the branching clause g and a form an atomic guarded action, and processes may interleave after evaluation of the guarded action at each loop iteration, and optionally in the non-atomic construct na, if one is present. Once again, each atomic clause maps to a guarded event. Conditional waiting is specified in a procedure using a conditional waiting construct of the following form,

$$\textbf{when}(\ g\ )\{\ a\ \}$$

where g is a guard and a is an action. A clause corresponds to the guarded action, $g \rightarrow a$, where g maps to an event guard, and a is mapped to an event action.

5 The Implementation-level Specification

The implementation level refinements are shown in Figures 3 and 4. We see that $StartWrite$ is implemented by clauses labelled $p1$ and $p2$; the iterating $WritePacket$ event is implemented by clauses $p3 \ldots p5$; and $p6$ implements

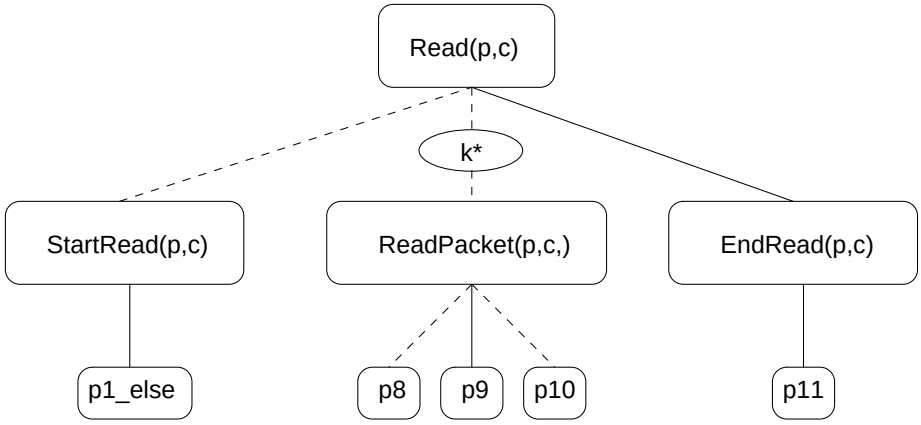

Figure 4: Implementing the Read Event

EndWrite. A similar arrangement exists for the reading process. A brief description of the clauses follows,

Label	Description
p1	If the process is a writer then get the size of the local buffer.
p2	Obtain the write channel if it is free else block.
	The process ID and number of packets to send are parameters
p3	While there is a packet to send from the local buffer
	remove the packet assigning to the temporary attribute.
p4	Add the data to the channel buffer.
p5	Decrement the count of packets.
p6	Release the channel for other writers.
p1_else	If the process is a reader obtain a read channel
	if it is free, else block.
p7	Obtain the number of packets to read from the channel.
p8	While there are packets remove a packet from the channel
	buffer and assign to the temporary attribute.
p9	Add the packet to the local buffer.
p10	Decrement the buffer counter.
p11	Free the read channel for another reader.

The clauses added at the implementation level add flow control information, and manipulate data, which also involves additional steps to store shared data in local attributes.

Our design decision is specify a single process to perform both reading and writing tasks. The specific behaviour of the process is determined by the boolean parameter $isWriter$ supplied at instantiation. The process class $Proc$ specification follows,

```
ProcessClass Proc{
 Buffer buff, Boolean isWriter, Channel c, Integer id,
 Integer tmpBuffSz, Integer tmpDat
 // The constructor procedure
 Procedure create(Integer pid, Buffer bff,
                        Boolean isWritr, Channel ch){
  id:=pid || buff:=bff || isWriter:=isWritr || c:= ch ||
  tmpBuffSz:=-1 || tmpDat:=-1
 }
 // The process behaviour
 Operation run(){
  p1: if(isWriter=TRUE) then
       tmpBuffSz:=buff.getSize() andthen
       p2: c.getWChan(id, tmpBuffSz);  // refines StartWrite
       p3: while(tmpBuffSz>0) do tmpDat:=buff.remove() andthen
            p4: c.add(tmpDat);  // refines WritePacket
            p5: tmpBuffSz:=tmpBuffSz-1 endwhile ;
       p6: c.freeWChan() endif  // refines EndWrite
  else c.getRChan(id) andthen  // refines StartRead
       p7: tmpBuffSz:=c.getWriteSize();
       p8: while(tmpBuffSz>0) do tmpDat:=c.remove() andthen
            p9: buff.add(tmpDat);  // refines ReadPacket
            p10: tmpBuffSz:=tmpBuffSz-1 endwhile ;
       p11: c.freeRChan() endelse  // refines EndRead
 }
}
```

We now look at the monitor class specification of the channel. *Channel* has a cyclic buffer $buff$ of, capacity 5; integer data elements are added to the tail and removed from the head of the buffer. A now provide an informal description of the monitor procedures,

add	Add a packet to the buffer tail, block the caller if there is no spare capacity.
remove	Remove a packet from the buffer head and return it, block the caller if there is nothing to remove.
getWChannel	Obtain a channel for writing if it is available and there is no reader, else block the caller.
$freeWChan$	Release a write channel by removing the process ID.
getRChannel	Obtain a channel for reading if it is available and there is data to read, else block the caller.
$freeRChan$	Release a read channel by removing the process ID.
$getWriteSize$	Returns the size of the data block.

The monitor procedures described above are specified in the Channel Monitor-

Class. The MonitorClass serves to encapsulate its attributes, access to data is only permissible through atomic procedure calls.

```
MonitorClass Channel{
 // Attributes
 Integer capacity, Integer[5] buff, Integer head, Integer tail,
 Integer size, Integer rPID, Integer wPID, Integer writeSize

 // The Constructor
 Procedure create(){
  head:= 0 || tail:= 0 || size:= 0 || capacity:= 5 ||
  rPID:= -1 || wPID:= -1 || writeSize:= -1
 }

 // 'Refines' WritePacket - in a call from clause p4
 Procedure add(Integer val){
  when(size<capacity & capacity /= 0 & tail>=0 & tail<= 4){
         buff[tail]:= val || tail:= (tail+1) mod capacity ||
         size:= size+1}
 }

 // The value is stored in a temporary buffer in a
 // call from clause p8 - implementing ReadPacket
 //as part of the reading activity.
 Procedure remove(){
  when(size>0  & capacity /= 0 & head>=0 & head<=4){
         return:= buff[head] || size:= size-1 ||
         head:= (head+1) mod capacity}
 }: Integer

 // Called in p1_else clause - refines StartRead.
 // Set the channel for reading, by the process
 // with identifier pid.
 // Block if it is already owned or has nothing to read.
 Procedure getRChan(Integer pid){
  when(rPID=-1 & writeSize>0){rPID:= pid}
 }

 // Called in p11 clause - refines EndRead.
 // Free the channel for reading.
 Procedure freeRChan(){
  rPID:= -1 || writeSize:= -1
 }

 // Called in p1 clause - implementing StartWrite.
```

```
 // Set the channel for writing writesze bytes, by
 // the process pid.
 // Block if the channel is already owned for writing or
 // has bytes still to write.
 Procedure getWChan(Integer pid,Integer writeSze){
   when(wPID=-1 & writeSize<=0){
          wPID:= pid || writeSize:= writeSze}
 }

 // Called in p6 clause - refines EndWrite.
 // Free the channel for writing.
 Procedure freeWChan(){ wPID:= -1 }

 // Return the number of bytes to write.
 Procedure getWriteSize(){ return:= writeSize }: Integer
}
```

A *MainClass* is used as the entry point for execution and is considered to be a special kind of process. The processes may equally be initiated by a GUI or scheduler.

6 The Implementation Refinement

The Event-B example shown here is the translation of the *Proc* class' write activity, specified in $p1 \ldots p6$

$Proc\_p1$ arises from the clause labelled $p1$,

$$
\begin{array}{l}
Proc\_p1 \triangleq \\
\quad \textbf{ANY } self,\ target \\
\quad \textbf{WHERE } self \in Proc \ \wedge \ self \in dom(Proc\_state) \wedge \\
\qquad Proc\_state(self) = p1 \ \wedge \ Proc\_isWriter(self) = TRUE \wedge \\
\qquad self \in dom(Proc\_buff) \ \wedge \ target = Proc\_buff(self) \\
\quad \textbf{THEN } Proc\_tmpBuffSz(self) := Buffer\_size(target) \parallel \\
\qquad Proc\_state(self) := p2 \\
\quad \textbf{END}
\end{array}
$$

$Proc\_p2$ refines $StartWrite$ of the first refinement. The $StartWrite$ process p is related to $self$ of $Proc\_p2$ using a predicate $p = self$ in the event's $WITH$ clause. The $StartWrite$ channel c is related to $target$ of $Proc\_p2$ with $c =$

$target$.

$Proc\_p2 \triangleq$
REFINES $StartWrite$
ANY $self,\ target$
WHERE $self \in Proc\ \wedge\ self \in dom(Proc\_state)\ \wedge$
$Proc\_state(self) = p2\ \wedge\ self \in dom(Proc\_c)\ \wedge$
$target = Proc\_c(self)\ \wedge\ Channel\_wPID(target) = -1\ \wedge$
$Channel\_writeSize(target) \leq 0$
THEN $Channel\_wPID(target) := Proc\_id(self)\ \wedge$
$Channel\_writeSize(target) := Proc\_tmpBuffSz(self)$
$Proc\_state(self) := p3$
END

$Proc\_p3$ is the start of the loop,

$Proc\_while\_p3 \triangleq$
ANY $self,\ target$
WHERE $self \in Proc\ \wedge\ self \in dom(Proc\_state)\ \wedge$
$Proc\_state(self) = p3\ \wedge\ Proc\_tmpBuffSz(self) > 0\ \wedge$
$self \in dom(Proc\_buff)\ \wedge\ target = Proc\_buff(self)\ \wedge$
$Buffer\_size(target) > 0\ \wedge\ Buffer\_capacity(target) \neq 0\ \wedge$
$Buffer\_head(target) \geq 0\ \wedge\ Buffer\_head(target) \leq 4$
THEN $Proc\_tmpDat(self) :=$
$Buffer\_buff(target)(Buffer\_head(target))\ \|$
$Buffer\_size(target) := Buffer\_size(target) - 1\ \|$
$Buffer\_head(target) :=$
$(Buffer\_head(target) + 1)\ mod\ Buffer\_capacity(target)\ \|$
$Proc\_state(self) := p4$
END

When proving the refinement of the implementation model we use gluing invariants to relate the abstraction with the implementation. An example of such an invariant follows,

$$\begin{aligned}&\forall pr, ch \cdot pr \in proc \wedge ch \in chan \wedge \\ &pr \in dom(Proc_c) \wedge Proc\_c(pr) = ch \wedge \\ &Proc\_id(pr) = Channel\_wPID(ch) \\ &\Rightarrow pr \mapsto ch \in writing\end{aligned}$$

This states that if a channel implementation has a writing process identifier value as its $wPID$ attribute (given by Proc_id(pr) = Channel_wPID(ch)) then this implies that the process-channel pair should be in the writing set in the abstraction. A similar invariant exists for the readers,

$$\begin{aligned}&\forall pr, ch \cdot pr \in proc \wedge ch \in chan \wedge \\ &pr \in dom(Proc_c) \wedge Proc\_c(pr) = ch \wedge \\ &Proc\_id(pr) = Channel\_rPID(ch) \\ &\Rightarrow pr \mapsto ch \in reading\end{aligned}$$

We also wish to ensure that no processes have the identifier -1, which is reserved for indicating that the channel has no reader/writer, We specify the following,

$$\forall pr \cdot pr \in Proc \wedge pr \in dom(Proc\_id) => Proc\_id(pr) \neq -1$$

7 The Java Implementation

The mapping to Java is mostly self evident since it is very similar to the OCB specification, we therefore present it without extensive explanation. The processes are implemented by threads that implement the Java *Runnable* interface.

```
public class Proc implements Runnable {

 private Buffer buff = null; private boolean isWriter;
 private  Channel c = null; private int id;
 private int tmpBuffSz; private int tmpDat;

 public Proc(int pid, Buffer bff, boolean isWritr, Channel ch) {
   id = pid; buff = bff; isWriter = isWritr; c = ch;
   tmpBuffSz = -1; tmpDat = -1;
 }

 public void run() {
   if (isWriter == true) {
     tmpBuffSz = buff.getSize(); // p1
     c.getWChan(id, tmpBuffSz); // p2
     while (tmpBuffSz > 0) {
       tmpDat = buff.remove(); // p3
       c.add(tmpDat); // p4
       tmpBuffSz = tmpBuffSz - 1; // p5
     }
     c.freeWChan(); // p6
   }else {
     c.getRChan(id);
     tmpBuffSz = c.getWriteSize(); // p7
```

```
    while (tmpBuffSz > 0) {
      tmpDat = c.remove(); // p8
      buff.add(tmpDat); // p9
      tmpBuffSz = tmpBuffSz - 1; // p10
    }
    c.freeRChan(); // p11
  }
 }
}
```

... and the Channel class is implemented and encapsulated by Java constructs. Atomic procedures are implemented by synchronized methods. The conditional wait construct is implemented by a $try - wait - catch$ block.

```
public class Channel {
 private int capacity;private int[] buff = new int[5];
 private int head;private int tail;private int size;
 private int rPID;private int wPID;private int writeSize;

 public Channel() {
  head = 0; tail = 0; size = 0; capacity = 5; rPID = -1;
  wPID = -1; writeSize = -1;
 }

 public synchronized void add(int val) {
  try {
    while (!(size < capacity && capacity != 0 &&
                              tail >= 0 && tail <= 4)) {
      wait();
  }} catch (InterruptedException e) { e.printStackTrace(); }
  buff[tail] = val;
  tail = (tail + 1) % capacity;
  size = size + 1;
  notifyAll();
 }

 public synchronized int remove() {
  int initial_head = head;
  try {
    while (!(size > 0 && capacity != 0 && head >= 0 && head <= 4)) {
      wait();
      initial_head = head;
    } catch (InterruptedException e) { e.printStackTrace();}
  size = size - 1;
  head = (initial_head + 1) % capacity;
  notifyAll();
```

```
    return buff[initial_head];
  }

  public synchronized void getRChan(int pid) {
    try {
    while (!(rPID == -1 && writeSize > 0)) {
      wait();
    }} catch (InterruptedException e) { e.printStackTrace(); }
    rPID = pid;
    notifyAll();
  }

  public synchronized void freeRChan() {
    rPID = -1;
    writeSize = -1;
    notifyAll();
  }

  public synchronized void getWChan(int pid, int writeSze) {
    try {
      while (!(wPID == -1 && writeSize <= 0)) {
        wait();
      }} catch (InterruptedException e) { e.printStackTrace(); }
    wPID = pid;
    writeSize = writeSze;
    notifyAll();
    }

  public synchronized void freeWChan() {
    wPID = -1;
    notifyAll();
  }

  public synchronized int getWriteSize() {
    return writeSize;
  }
}
```

8 Conclusion

In this example we developed an Event-B model of reading and writing processes sharing a channel. In topmost abstraction we modelled reads and writes of blocks of data, this was refined so that reads and writes of individual packets of data were modelled. We gave an overview of the OCB notation, which we use to specify implementations for concurrent processes sharing data. We also gave

an overview of JSDs and showed how they can be beneficial in understanding the relationship between abstract and refined events, and event ordering, as a development proceeds. The OCB notation was used to provide an implementation level specification for our abstract development, which was subsequently mapped to Java code. We also map to an Event-B model which models the implementation, and are required to show that this implementation model is a refinement of the abstract development (on-going work).

References

[1] Michael Butler. Decomposition Structures for Event-B. In *Integrated Formal Methods iFM2009, Springer, LNCS 5423*, volume LNCS. Springer, February 2009.

[2] A. Edmunds and M. Butler. Linking Event-B and Concurrent Object-Oriented Programs. In *Refine 2008 - International Refinement Workshop*, May 2008.