ELECTRONIC **WORKSHOPS IN COMPUTING**

Series edited by Professor C.J. van Rijsbergen

D Duke, University of York, UK and A S Evans, University of Bradford, UK (Eds)

BCS-FACS Northern Formal Methods Workshop

Proceedings of the BCS-FACS Northern Formal Methods Workshop, Ilkley, UK. 23-24 September 1996

Analysis of the Four-slot Mechanism

P.J. Brooke, J.L. Jacob and J.M. Armstrong

Published in collaboration with the British Computer Society

BCS FACS

Springer

Analysis of the Four-slot Mechanism

Phillip J. Brooke
Corresponding author, <pjb@minster.york.ac.uk>
Department of Computer Science, The University of York
York, YO1 5DD, England

Jeremy L. Jacob
Department of Computer Science, The University of York
York, YO1 5DD, England

James M. Armstrong
Department of Computer Science, The University of York
York, YO1 5DD, England

Abstract

This paper presents an analysis of an asynchronous communication mechanism due to Simpson [4]. The properties required of an abstract version of the mechanism are discussed. Simpson's mechanism is described in terms of a process algebraic model. We then use automated techniques to test this mechanism against the properties stated.

1 Introduction

In concurrent systems, techniques for preventing conflict in reader-writer communication are usually based on mutual exclusion principles [2].

By contrast, Simpson suggests a *"fully asynchronous communication mechanism"* [4]. It is intended to provide one reader and one writer with an efficient communication mechanism. By the use of four *data slots* and four additional *control bits* to control the routing of data to and from the slots, it is claimed that the system possesses three properties: *asynchrony*, *coherence*, and *freshness*.

Asynchrony The first of Simpson's requirements is:

> "*neither process may affect the timing of the other as a direct result of its communication operations.*" [4]

Coherence Simpson also requires that:

> *"data must always be passed as a coherent set, i.e. interleaved access to any [data slot] by the writer and the reader is not permitted.*" [4]

Freshness In his 1990 paper, Simpson required that

> *"the latest complete data record produced by the writer must always be made available for use by the reader.*" [4]

However, his 1992 paper requires that,

> "*the data obtained by the reader is always the most recent (freshest) to have been supplied by the writer.*" [5]

$$
\begin{aligned}
SW &= \{ d : D \bullet sw.d \} \\
ER &= \{ d : D \bullet er.d \} \\
\alpha POOL &= \{sr, ew\} \cup SW \cup ER
\end{aligned}
$$

Figure 1: Alphabet of $POOL$

Implicitly, old data may be overwritten by newer data (if the writer is faster than the reader) and data may be read repeatedly (if the reader is faster than the writer).

Structure of this paper

In Section 2 we discuss the formalisation of Simpson's properties, above. Then, in Section 3, we present a model of the four-slot mechanism. In both cases we uses Hoare's CSP [3]; in the former case we use predicates about failures while in the latter case we use the process algebra.

An analysis of the mechanism against the specification is given in Section 4 and some conclusions are offered in Section 5.

2 Black Box Model

We suppose a writer is sending data to a reader via some intermediary mechanism, which we will call '$POOL$'. Interactions with the mechanism, such as 'write a datum' and 'read a datum' are modelled as a pair of events, one denoting the start of the interaction, and one denoting the end of the interaction. This allows us to reason explicitly about overlapping reads and writes.

The alphabet of this mechanism —the atomic events it can synchronise on— are given in Figure 1. The event $sw.d$ denotes the start of a write, and ew denotes that the write has ended, and that the value $d \in D$ has been written. Similarly, sr and $er.d$ denote the start and end of a read of value d.

It is an important part of our modelling that the datum transferred is fixed at the start of a write and the end of a read.

2.1 Specification Style

We now give predicates which $POOL$ is required to satisfy. These predicates are failure specifications, in the style of Hoare's CSP [3].

The pair (s, X) is a *failure* of a process P, exactly when the process P may engage in the sequence of events s (s is called a *trace* of P), and then refuse to do any more, even though the environment is prepared to engage in any event in X (X is called the *refusal set* of the failure; X is called a *refusal set* of the process P if $(\langle\rangle, X)$ is a failure of P).

A specification of a process is a (collection of) predicate(s) limiting the failures the process may exhibit.

In the predicates, 'tr' is a free variable representing a typical trace and 'rf' a free variable representing a refusal set coupled with tr in a failure (tr, rf). Other notations used in this paper are:

$s \# A$ means the number of occurrences of elements of the set A in the sequence s

$s \upharpoonright A$ denotes the restriction of the sequence s to members of the set A

$front\ s$ returns s without its last element, where s is a nonempty sequence; returns the empty sequence otherwise

$last\ s$ returns the last element of the nonempty sequence s

$$
\begin{aligned}
READING &= (tr\#\{sr\} - tr\#ER = 1) \\
NOTREADING &= (tr\#\{sr\} - tr\#ER = 0) \\
WRITING &= (tr\#SW - tr\#\{ew\} = 1) \\
NOTWRITING &= (tr\#SW - tr\#\{ew\} = 0) \\
ALTERNATE &= (READING \lor NOTREADING) \\
&\quad \land (WRITING \lor NOTWRITING)
\end{aligned}
$$

Figure 2: Definition of $ALTERNATE$

$$
\begin{aligned}
ASYNCH = \quad & NOTREADING \Rightarrow sr \notin rf \land READING \Rightarrow ER \setminus rf \neq \emptyset \\
& \land NOTWRITING \Rightarrow SW \cap rf = \emptyset \land WRITING \Rightarrow ew \notin rf
\end{aligned}
$$

Figure 3: Definition of $ASYNCH$

2.2 Alternation

It is reasonable to expect that the writer engages in start and end events alternately (and similarly for the reader). The predicate $ALTERNATE$, presented in Figure 2, captures this requirement. This is defined in terms of four further predicates, $READING$, $NOTREADING$, $WRITING$ and $NOTWRITING$.

2.3 Asynchrony

It is required that

- The reader may start to read if it is not already reading.
- The reader may end a read if it is already reading.
- The writer may start to write if it is not already writing.
- The writer may end a write if it is already writing.

This is easily specified by limiting the process's capacity to refuse. The relevant predicate, $ASYNCH$, is presented in Figure 3. $ASYNCH$ says:

- When a read is not in progress the process may not refuse to allow a read to start,
- when a read is in progress the process may not refuse to allow the read to end,
- when a write is not in progress the process may not refuse to allow a write to start, and
- when a write is in progress the process may not refuse to allow the write to end.

The phrase "may not refuse to allow" is, here, equivalent to the phrase "must allow", but is a more literal rendering of the predicate.

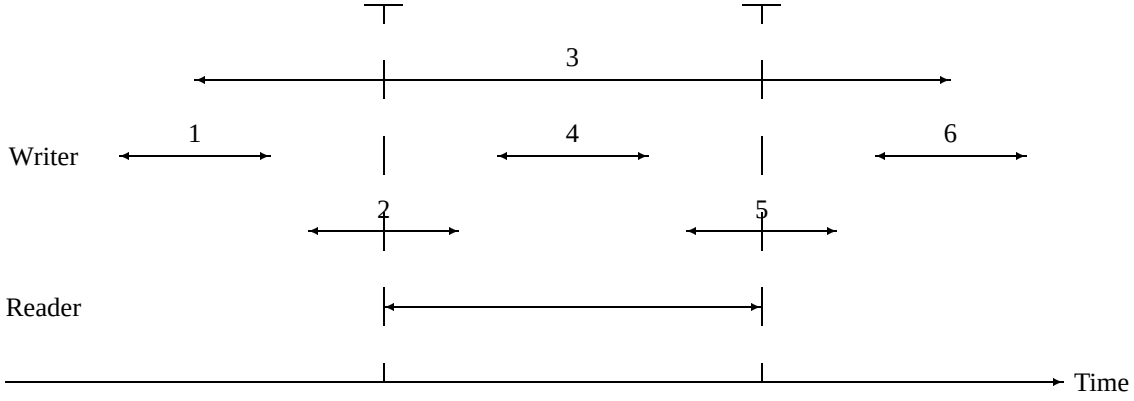

Figure 4: What is a fresh read?

$$FRESH = \forall d : D \bullet last\ tr = er.d \Rightarrow RECENT(d)$$

Figure 5: Definition of $FRESH$

2.4 Coherence

Coherence, as Simpson defines it (see Section 1), is a property that is not directly visible outside the mechanism. Hence we do not give predicates for it in terms of external events, and instead model the failure of coherence when we describe the four-slot mechanism itself.

It does have implications for the observable behaviour, however, but these are dealt with under the head of "Freshness", below (Section 2.5).

2.5 Freshness

Freshness is a tricky property to tie down. A full analysis probably needs a notion of real time, and the use of a model such as Timed CSP [1].

Simpson knows that his mechanism allows a just-read datum to be other than the most recently written datum [5]. His belief is that, at worst, the second most recent datum is read. Later (Section 4.4) we show that this belief is not justified.

Some problems with the definition of freshness arise because it is not clear what is meant by 'this read occurs after this write'. Consider Figure 4, which shows all ways in which reads and writes can overlap. Should we take the starts or ends of actions, or a mixture?

Our solution is to define the predicate $FRESH$. The definition is given in Figure 5, and is in terms of another predicate $RECENT(d)$. $RECENT(d)$ expresses the fact that datum 'd' has been recently written.

Four variants of the latter predicate are given, each differing in the way it compares the starts and ends of reads and writes. These variants are presented in Figure 7 (conjoining comparisons 1,2 and 3 of Figure 4), Figure 8 (comparison 1 of Figure 4), Figure 9 (comparisons 1–5 of Figure 4), and Figure 10 (comparisons 1–4 of Figure 4).

Each version of $RECENT(d)$ is in terms of a predicate $WRC(X, d, s)$ ('write-read comparison'; see Figure 6) that gives the part of the write the read is compared against. The first part of the disjunction in $WRC(X, d, s)$ allows the default value d_0 to be read if there has been at most one write. The second disjunct allows for the case when only one write has occurred in the trace. The third disjunct permits the second most fresh data to be returned.

Data that is not fresh (i.e. the 'freshest' or 'previous freshest') is called *stale*.

$$\begin{aligned} WRC(X,d,s) \quad = \quad & (s\#X \leq 1 \wedge d_0 = d) \\ & \vee\ (s\#X \geq 1 \wedge last(s \upharpoonright SW) = sw.d) \\ & \vee\ (s\#X \geq 2 \wedge last(front(s \upharpoonright SW)) = sw.d) \end{aligned}$$

Figure 6: Definition of $WRC(X,d)$

$$RECENT(d) = tr = s\hat{}\langle sr\rangle\hat{}t \wedge \neg(\langle sr\rangle \textbf{ in } t) \wedge WRC(SW,d,s)$$

Figure 7: Definition of $RECENT(d)$, comparing start-writes with start-reads

$$RECENT(d) = WRC(SW,d,tr)$$

Figure 8: Definition of $RECENT(d)$, comparing start-writes with end-reads

$$RECENT(d) = tr = s\hat{}\langle sr\rangle\hat{}t \wedge \neg(\langle sr\rangle \textbf{ in } t) \wedge WRC(\{ew\},d,s)$$

Figure 9: Definition of $RECENT(d)$, comparing end-writes with start-reads

$$RECENT(d) = WRC(\{ew\},d,tr)$$

Figure 10: Definition of $RECENT(d)$, comparing end-writes with end-reads

```
mechanism four slot;

  var
     data : array [bit,bit] of datatype := ((null,null),(null,null));
     slot : array [bit] of bit := (0,0);
     latest, reading : bit := 0;

  procedure write (item : datatype);
    var pair,index : bit;
  begin
    pair := not reading;
    index := not slot[pair];
    data[pair,index] := item;
    slot[pair] := index;
    latest := pair
  end write;

  function read : datatype;
    var pair,index : bit;
  begin
    pair := latest;
    reading := pair;
    index := slot[pair];
    return data[pair,index]
  end read;

end four slot.
```

Figure 11: The four-slot mechanism in pseudo-code form

3 Four-slot Model

We now discuss Simpson's mechanism, which is presented in Figure 11 in its original pseudo-code form, with minor typographic changes. 'data[$\cdot$,$\cdot$]' holds the actual data. 'slot[$\cdot$]' and 'latest' are only written by procedure write, and reading is only written by procedure read.

Figure 12 is a block diagram of the mechanism. The mechanism consists of the data slots and control bits mentioned above, and a protocol. The protocol is represented by the processes $WRITER$ and $READER$.

A process writes by sending the data to the $WRITER$ component of the mechanism. This communicates with the $READER$ component (and hence the reading process) via four data slots ($data\ldots$), and four control bits (comprising: $reading$, $latest$, and two $slot$ bits). These components correspond to the variables and algorithms in Simpson's pseudo-code.

We now define each component in turn, and describe the assumptions underlying our model.

Data slots As for external reads and writes, interactions with the data slots are modelled as two events, one denoting the start of the interaction, and one denoting the end of the interaction.

We define a process DS to model the internal data slots. The event sw denotes the start of a write, and $ew.d$ denotes that the write has ended, and that the value $d \in D$ has been written. Similarly, sr and $er.d$ denote the start and end of a read of value d.

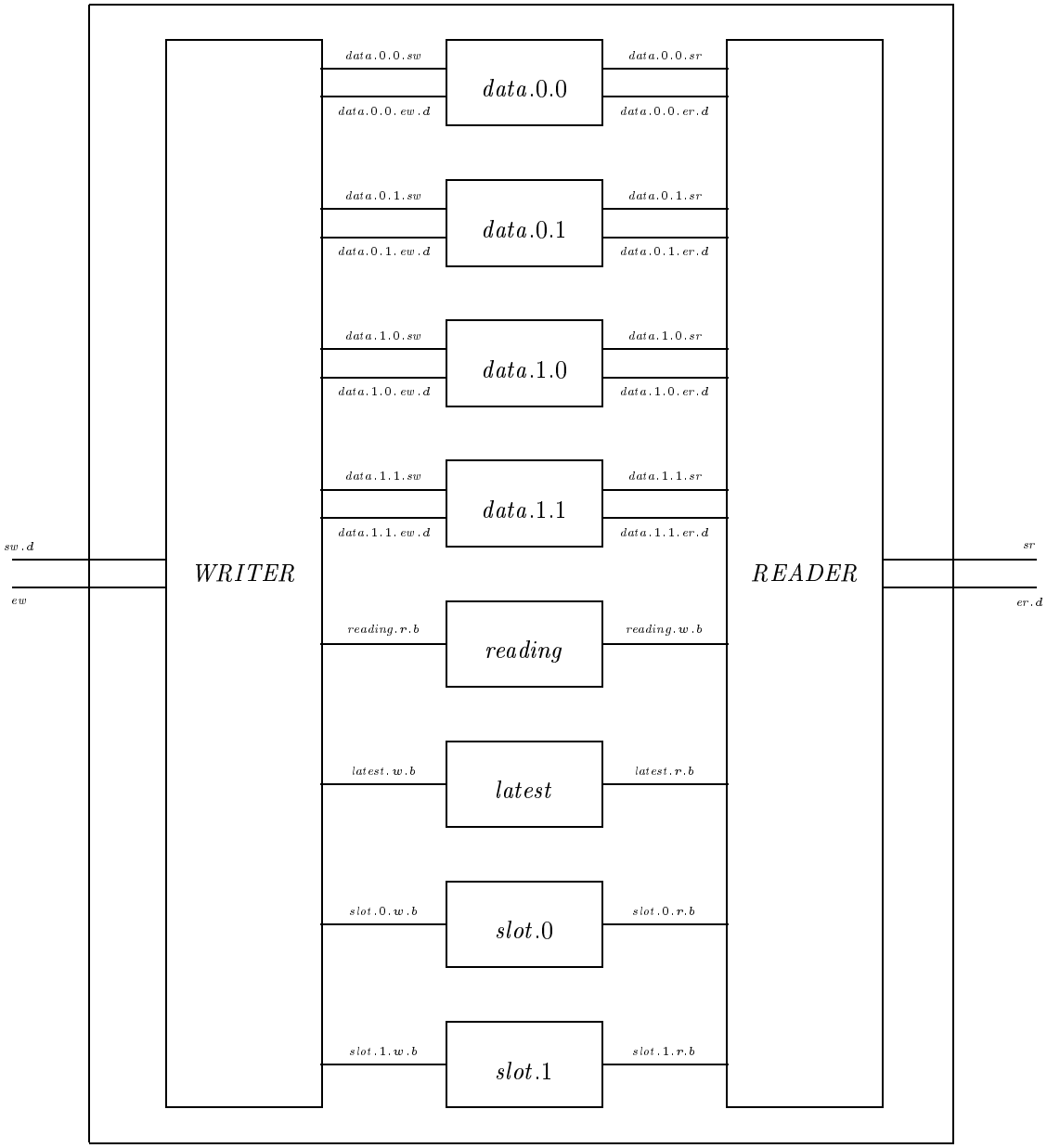

Figure 12: Block diagram of the four-slot model

$$
\begin{aligned}
\alpha DS &= \{sr, sw\} \cup \{\, d : D \bullet er.d \,\} \cup \{\, d : D \bullet ew.d \,\} \\
DS &= S(0, d_0) \\
S(0, d) &= sr \rightarrow S(1, d) \,\Box\, sw \rightarrow W \\
S(i+1, d) &= sr \rightarrow S(i+2, d) \,\Box\, er!d \rightarrow S(i, d) \,\Box\, sw \rightarrow COLLISION \\
W &= ew?d : D \rightarrow S(0, d) \,\Box\, sr \rightarrow COLLISION \,\Box\, sw \rightarrow COLLISION \\
COLLISION &= collide \rightarrow STOP
\end{aligned}
$$

Figure 13: Definition of DS

$$
\begin{aligned}
\alpha BIT &= \{w.0, w.1, r.0, r.1\} \\
BIT &= BIT(0) \\
BIT(y) &= w?x \rightarrow BIT(x) \,\Box\, r!y \rightarrow BIT(y)
\end{aligned}
$$

Figure 14: Definition of BIT

Note that this definition allows multiple readers and multiple writers. However, a collision (two processes carrying out overlapping operations) between two writers, or between a reader and a writer will result in the $COLLISION$ process. This is pessimistic; there is no recovery from error. If the protocol works with these slots, then it will work with real slots. This discharges our requirements for modelling coherence.

Figure 13 describes the process DS, where d_0 is the initial value held by the DS.

Control bits We define a process BIT in Figure 14. These interactions are considered to be atomic, so we model them as single events.

We define the mechanism comprising the data slots and control bits in Figure 15.

Writer This process engages in the two external events: $sw.d$ and ew. See Figure 16. B is defined as $\{0, 1\}$, and $\bar{b}$ denotes $1 - b$ for $b \in B$.

Reader Similarly, the reader can engage in the external events sr and $er.d$ (Figure 17).

Mechanism We define the overall mechanism, $FOURSLOT$, in Figure 18. The only events visible outside the mechanism are sr, $er.d, d \in D$, $sw.d, d \in D$ and ew.

Simpson has described a wide range of variants; some return pointers to the data slots ('data-visibility mechanisms'). Others have only 'read pre-sequences' and write 'post-sequences' [4, 5].

What we would like to test is the claim that $FOURSLOT$ satisfies each of $ASYNCH$, $ALTERNATE$, and each variety of $FRESH$.

4 Analysis of the Four-slot Model

4.1 A Model Checker

Hand analysis of a mechanism even as small as $FOURSLOT$ is tedious. We performed an analysis using several tools, as described below.

$$
\begin{aligned}
SLOTS &= data.0.0 : DS \parallel data.0.1 : DS \parallel data.1.0 : DS \parallel data.1.1 : DS \\
BITS &= reading : BIT \parallel latest : BIT \parallel slot.0 : BIT \parallel slot.1 : BIT \\
VARS &= SLOTS \parallel BITS
\end{aligned}
$$

Figure 15: Data slot and component definition

$$
\begin{aligned}
\alpha WRITER = \quad & \{d : D \bullet sw.d\} \cup \{ew\} \\
& \cup \{p : B \bullet reading.r.p\} \cup \{p : B \bullet latest.w.p\} \\
& \cup \{p, i : B \bullet slot.p.r.i\} \cup \{p, i : B \bullet slot.p.w.i\} \\
& \cup \{p, i : B \bullet data.p.i.sw\} \cup \{p, i : B; d : D \bullet data.p.i.ew.d\} \\
WRITER = \quad & sw?d \\
& \rightarrow reading.r?p \\
& \rightarrow slot.\bar{p}.r?i \\
& \rightarrow data.\bar{p}.\bar{i}.sw \\
& \rightarrow data.\bar{p}.\bar{i}.ew!d \\
& \rightarrow slot.\bar{p}.w!\bar{i} \\
& \rightarrow latest.w!\bar{p} \\
& \rightarrow ew \\
& \rightarrow WRITER
\end{aligned}
$$

Figure 16: Definition of $WRITER$

$$
\begin{aligned}
\alpha READER = \quad & \{sr\} \cup \{ d : D \bullet er.d \} \\
& \cup \{ p : B \bullet latest.r.p \} \cup \{ p : B \bullet reading.w.p \} \\
& \cup \{ p, i : B \bullet slot.p.r.i \} \\
& \cup \{ p, i : B \bullet data.p.i.sr \} \cup \{ p, i : B; d : D \bullet data.p.i.er.d \} \\
READER = \quad & sr \\
& \rightarrow latest.r?p \\
& \rightarrow reading.w!p \\
& \rightarrow slot.p.r?i \\
& \rightarrow data.p.i.sr \\
& \rightarrow data.p.i.er?d \\
& \rightarrow er!d \\
& \rightarrow READER
\end{aligned}
$$

Figure 17: Definition of $READER$

$$
\begin{aligned}
INTERNALS &= WRITER \parallel READER \parallel VARS \\
FOURSLOT &= INTERNALS \backslash (\alpha INTERNALS \setminus (\{sr, ew\} \cup SW \cup ER))
\end{aligned}
$$

Figure 18: Definition of $FOURSLOT$

$$
\begin{aligned}
FDRASYNCREADER &= sr \rightarrow \left(\bigsqcap d : D \bullet er!d \rightarrow FDRASYNCREADER \right) \\
FDRASYNCWRITER &= sw?d \rightarrow ew \rightarrow FDRASYNCWRITER \\
FDRASYNCH &= FDRASYNCREADER \parallel FDRASYNCWRITER
\end{aligned}
$$

Figure 19: FDR alternation and asynchrony specification

We built our own, problem-specific tool, a state-space checker. This program constructed a directed graph for the four-slot model, where each node represented a state, and each edge an event engaged in by either the reader or the writer.

Initially, only one node is in the graph: the node which corresponds to the system when no events have been engaged in. The following algorithm is then executed.

For each unprocessed node in the graph
  Check if the writer can engage in an event
    Insert an edge for 'writer engages in event',
    and if necessary, create a node
  (Similarly for the reader)

Each time an edge is inserted, the node that it is connected to is determined: if this node is already in the graph, then nothing further needs doing. If the node is not in the graph, then it is created. The number of nodes in the graph was reduced by determining which reader and writer values and bits (d, p and i) 'don't matter' (when a given bit value will not be referenced again in that cycle of events), and coalescing such nodes together. The size of the graph was relatively small when the data type stored by the data slots, DS, was the empty type (1512 nodes), and as expected, Boolean data resulted in a large increase in the number of states (40000–180000 nodes, depending on the model).

We also used the Failures-Divergences Refinement (FDR) tool to test specifications by constructing the most general process that satisfies the specification. FDR then checks whether our model (the implementation) refines this process.

We now summarise the results from both tools.

4.2 Alternation and Asynchrony

In the case of the model checker, for each node, there was exactly one event available to the reader, and at least one node for the writer. This shows that the asynchrony specification is satisfied by the mechanism: both the reader and the writer can make progress in every state.

The FDR model was checked against the specification in Figure 19.

The $FOURSLOT$ model refined $FDRASYNCH$.

4.3 Coherence

Both tools were unable to find a state where a collision had occurred.

The model checker checked which data slots (if any) the reader and writer were operating on in each state, and found no states where the reader and writer were operating on the same data slot.

External events		Internal events			
Writer	Reader	Writer	Reader	Previous	Fresh
				0	0
	sr			0	0
$sw.1$					
		$reading$.r.0			
		$slot$.1.r.0			
		$data$.1.1.sw			
		$data$.1.1.ew.1			
		$slot$.1.w.1			
		$latest$.w.1			
			$latest$.r.1		
			$reading$.w.1		
			$slot$.1.r.1		
			$data$.1.1.sr		
			$data$.1.1.er.1		
	$er.1$			0	0

Figure 20: A stale-read (start-write/start-read)

External events		Internal events			
Writer	Reader	Writer	Reader	Previous	Fresh
				0	0
$sw.1$				0	1
		$reading$.r.0			
		$slot$.1.r.0			
		$data$.1.1.sw			
		$data$.1.1.ew.1			
		$slot$.1.w.1			
	sr				
			$latest$.r.0		
		$latest$.w.1			
ew					
$sw.1$				1	1
			$reading$.w.0		
			$slot$.0.r.0		
			$data$.0.0.sr		
			$data$.0.0.er		
	$er.0$			1	1

Figure 21: A stale-read (start-write/end-read)

The process used as a specification for FDR was

$$RUN(\alpha(INTERNALS) \setminus \{collide\})$$

which will engage in anything at any time except that it may never engage in the $collide$ event. This was used to test $INTERNALS$ rather than $FOURSLOT$ (since we were interested in the internal workings of the mechanism). FDR found that $INTERNALS$ refined the process under the traces model. (We are only interested in the safety property here that $collide$ is never engaged in; we do not need the failures model here.)

This demonstrates that the mechanism is coherent in Simpson's original sense.

4.4 Freshness

When we explored the graph produced by the model checker, with the data slots holding Boolean values, we found ways of obtaining a stale datum. This was true for each of the different notions of "recently written", $RECENT(d)$, discussed above. Figures 20, 21, 22 and 23 show examples of traces which lead to reads of stale data. Attempts to use a data type with more than two values failed due to the large amount of time and computer resources required (partially due to the inefficient program).

External events		Internal events			
Writer	Reader	Writer	Reader	Previous	Fresh
				0	0
$sw.1$					
		$reading$.r.0			
		$slot$.1.r.0			
		$data$.1.1.sw			
		$data$.1.1.ew.1			
		$slot$.1.w.1			
		$latest$.w.1			
	sr			0	0
			$latest$.r.1		
			$reading$.w.1		
			$slot$.1.r.1		
			$data$.1.1.sr		
			$data$.1.1.er.1		
	$er.1$			0	0

Figure 22: A stale-read (end-write/start-read)

External events		Internal events			
Writer	Reader	Writer	Reader	Previous	Fresh
				0	0
$sw.1$					
		$reading$.r.0			
		$slot$.1.r.0			
		$data$.1.1.sw			
		$data$.1.1.ew.1			
		$slot$.1.w.1			
		$latest$.w.1			
	sr				
			$latest$.r.1		
			$reading$.w.1		
			$slot$.1.r.1		
			$data$.1.1.sr		
			$data$.1.1.er.1		
	$er.1$			0	0

Figure 23: A stale-read (end-write/end-read)

$$
\begin{aligned}
FDRFSWSR1 &= FDRFSWSR2(0,0,0,0) \\
FDRFSWSR2(rp, rf, wp, wf) &= sr \rightarrow FDRFSWSR3(wp, wf, wp, wf) \\
&\quad \Box\ sw?d \rightarrow FDRFSWSR4(rp, rf, wf, d, d) \\
FDRFSWSR3(rp, rf, wp, wf) &= er!rf \rightarrow FDRFSWSR2(rp, rf, wp, wf) \\
&\quad \Box\ er!rp \rightarrow FDRFSWSR2(rp, rf, wp, wf) \\
&\quad \Box\ sw?d \rightarrow FDRFSWSR5(rp, rf, wf, d, d) \\
FDRFSWSR4(rp, rf, wp, wf, d) &= sr \rightarrow FDRFSWSR5(wp, wf, wp, wf, d) \\
&\quad \Box\ ew \rightarrow FDRFSWSR2(rp, rf, wp, wf) \\
FDRFSWSR5(rp, rf, wp, wf, d) &= er!rf \rightarrow FDRFSWSR4(rp, rf, wp, wf, d) \\
&\quad \Box\ er!rp \rightarrow FDRFSWSR4(rp, rf, wp, wf, d) \\
&\quad \Box\ ew \rightarrow FDRFSWSR3(rp, rf, wp, wf)
\end{aligned}
$$

Figure 24: FDR process for checking start write-start read freshness

When using FDR, we devised processes for each of the four variants of freshness. The process for the start write-start read variant, $FDRFSWSR$, is defined in Figure 24. (We omit the processes for the remaining three variants for reasons of brevity.)

We discovered the same result as above in all four cases.

5 Conclusions

Simpson's four-slot mechanism satisfies the given requirements for asynchrony and coherence, but does not satisfy freshness, as given here. This result was determined by a model checking program written specifically for this investigation, and confirmed with FDR.

Simplistic attempts to adjust the freshness predicate fail: for example, we may allow the reader to return any of the three most recent reads. The trace in Figure 23 can easily be extended to demonstrate a lack of this more lax freshness. Note that these failed traces generally rely on a relatively long sequence from the writer or reader uninterrupted by the other process. In many circumstances, this is unrealistic. (However, Simpson did claim that there should be no timing constraints.)

This leads us to consider exactly what property we are trying to describe when discussing 'freshness'. Exactly when is data fresh? Do we need to consider the start and end of writes and reads in some relatively complicated expression? Is it particularly important in all circumstances — perhaps reading a value that has at some time in the past been written is acceptable. Freshness may be something more vague; data may more usefully become 'fresher' as the write progresses.

Alternatively, we could consider that the black box model of the mechanism is flawed. The stale traces presented generally rely on interleaving, say, internal reader events between the last writer internal event and its (black box) external event. A formulation where the external events are 'fixed' to the first and last internal events respectively may give more realistic results.

However, this then defeats the objective of determining abstract properties about the mechanism as a whole. Ideally, we would like to use the mechanism without having to reason about its internal structure. (Preliminary investigation of this 'fixing' suggests that stale traces can still be found unless most of the events are fixed to each other.)

The addition of timing to the model would broaden the range of plausible behaviours. (An extensive body of work exists on Timed CSP [1].) We may insist that any read that starts t time units after a write has ended must return the value of that write, or a subsequent write. However, the addition of timing to the model may render automated checking difficult.

We could also consider other behaviours that some component should satisfy. For example, should there be some notion of order? (e.g. If the values 0 then 1 are written, 1 then 0 should not be read.) This could be examined in the future, with other possible behaviours. Other useful predicates are important: the four-slot mechanism is an implementation of the *pool* in Simpson's DORIS methodology [6]. There are a number of broad types of *route* in DORIS, of which the pool is only one. So a general method for specifying types of route, and demonstrating their successful implementation would be useful.

Acknowledgements

This work has been financially supported by EPSRC and British Aerospace. The authors would like to thank Dr Hugo Simpson and Dr Steve Paynter of British Aerospace, and Nick Cropper of The University of York for their comments on this work.

References

[1] Davies J, Schneider S. A brief history of Timed CSP. Technical Report PRG-96, Programming Research Group University of Oxford, April 1992

[2] Dijkstra EW. Solution of a problem in concurrent programming control. Communications of the ACM 1965; 8(9):569

[3] Hoare CAR. Communicating Sequential Processes, Prentice-Hall International UK, 1985

[4] Simpson HR. Four-slot fully asynchronous communication mechanism. IEE Proceedings 1990; 137 Part E(1):17–30

[5] Simpson HR. Correctness analysis for class of asynchronous communication mechanisms. IEE Proceedings 1992; 139 Part E(1):35–49

[6] Simpson HR. Methodological and Notational Conventions in DORIS Real Time Networks. Dynamics Division, British Aerospace, 1994