School of Computing Science,
University of Newcastle upon Tyne

# Ramifications of Metastability in Bit Variables Explored via Simpson's 4-Slot Mechanism

S. E. Paynter, N. Henderson, J. M. Armstrong

Technical TR Series

CS-TR-789

24th January 2003

Contact:
stephen.paynter@mbda.co.uk
neil.henderson@ncl.ac.uk
j.m.armstrong@ncl.ac.uk


Published by the University of Newcastle upon Tyne,
School of Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, UK

# Ramifications of Metastability in Bit Variables Explored via Simpson's 4-Slot Mechanism

S. E. Paynter, N. Henderson, J. M. Armstrong

24th January 2003

**Abstract**

Protocol descriptions often fail to take metastability into account. Metastability, however, can undermine protocols which depend on shared bits. In this paper, a series of increasingly realistic models of bits are developed in CSP, to explore the implications of metastability for Simpson's 4-Slot asynchronous communication mechanism. It is shown that the 4-Slot mechanism with realistic bit models preserves data-coherence, freshness, and sequencing, and is Lamport-atomic. We demonstrate that metastability can undermine the correctness of protocols demonstrated correct with Lamport-safe models of bits; furthermore, realistic bit models can demonstrate protocols correct which Lamport-safe bit models would suggest were incorrect.

**Keywords:** Bits, Metastability, Simpson's 4-Slot Mechanism, Asynchronous Communication Mechanisms, Freshness

## 1 Introduction

Asynchronous communication occurs in modern digital electronic systems whenever a signal is passed between two circuits which do not share a common clock, for example, when one processor sets an interrupt for another. It is well known by electronic engineers involved in designing such systems that attempts to read an asynchronous message by the system receiving it may result in an ineliminable phenomenon called "metastability" in the receiving system. Unfortunately, metastability has a number of undesirable effects, some of which are discussed below. However, although metastable behaviour cannot be totally eliminated, digital circuits can be constructed to operate in ways which reduce the probability of problematic metastable behaviour to acceptably low levels. Hence, it is necessary to appreciate that these operating constraints are present when constructing formal models of systems which engage in asynchronous communication. The failure to recognise these constraints undermines the relevance of any analysis performed on such models.

At an abstract level, devices which support asynchronous communication are called either *asynchronous communication mechanisms* (ACMs) or *wait-free protocols*. Conceptually, ACMs support the communication of data between writing and reading processes which are unconstrained in when and at what rate they can access the mechanism. Not only may reads and writes overlap, but multiple consecutive writes may overlap a read and vice versa. ACMs are essentially *shared-variables*, and hence have the properties that:

- a value written into one may be read many times; and
- writing a value conceptually destroys (makes unavailable for reading) values written previously[1].

[1]The asynchronous communication that ACMs support therefore needs to be distinguished from the "asynchronous communication" supported by (infinite) buffers, for example, [JHJ89].

Shared-variables that are implemented as ACMs therefore have fundamentally different synchronisation properties from shared-variables that are implemented as Hoare-atomic (H-atomic, in the sequel) "monitors". Monitors use mutual exclusion to ensure that corrupted data cannot be read, [Hoa74].

ACMs are of particular interest for a number of reasons, including:

- They are present whenever systems that do not share a clock must communicate. This is even true when there is apparent support for synchronous communication. Such mechanisms need to be built from ACMs, although this may be at the hardware level and hidden from the software.
- Even more than other shared-variables, they support the integration of sub-systems or processes which run at different frequencies, or which are sporadic.
- They provide a means of decoupling the temporal interactions between systems that communicate. By definition, no process accessing an ACM can hinder another one from also accessing it.
- They provide a means of building systems which are robust against deadlock due to the failure of one of the communicating systems.

In two seminal papers in 1986, [Lam86a, Lam86b], Leslie Lamport introduced a classification scheme for ACMs or "asynchronous registers" that consisted of a cumulative hierarchy of properties called *safe*, *regular*, and *atomic*. These properties are prefixed by "L-" (for Lamport) in this paper for consistency, and to help to make a systematic distinction between the Lamport-atomic and Hoare-atomic properties.

*L-safe* ACMs guarantee that a non-clashing read (that is, a read that is not contemporaneous with a write) gets the value stored in the device, while a read that clashes with a write may return any valid value of the type being communicated. An *L-regular* ACM is an L-safe ACM that additionally guarantees that a clashing read gets either (a) the value of the immediately prior write which did not clash with that read, or (b) one of the values written by a write that clashes with that read. An *L-atomic* ACM is an L-regular ACM which ensures that clashing reads do not read earlier values than have already been read. This is equivalent to saying that the values returned by the reads are as if the reads and writes had occurred in some determinate non-clashing order. L-atomic ACMs therefore behave "like" H-atomic variables, but without the synchronisation effects arising from mutual exclusion.

This classification scheme has influenced much of the discussion and modelling of ACMs since 1986. However, Lamport's papers fail to discuss metastability explicitly, and therefore his classification scheme needs to be used with care. We demonstrate that metastability can undermine the correctness of protocols demonstrated correct with Lamport-safe models of bits; furthermore, that realistic bit models can demonstrate protocols correct which Lamport-safe bit models would suggest were incorrect.

Three properties that an ideal ACM should exhibit are *data-coherence*, *freshness*, and *sequencing*. Data-coherence is a vital property; it requires that values written into and read out of an ACM should not be corrupted. This can be made more precise by saying that every value read out will be one that was written in (or will be the initial one). Note, this is the definition of a *semi-regular* ACM that the first two authors identified as missing from Lamport's classification scheme in [HP02]. In some ACMs (such as the 4-Slot mechanism discussed below), data-coherence can be defined in terms of internal memory locations; the reader and writer should never concurrently access the same (data) memory location.

Freshness is a more difficult property to define precisely (see Subsection 6.3 below), however, intuitively it means that the reader should not obtain data older than the most recent data "available"[2]. Arguably, either the L-regular or the L-atomic ACM is the first ACM in Lamport's classification hierarchy to exhibit "freshness" – depending upon how freshness is defined.

[2]The subtleties arise in deciding what "available" means when multiple writes can occur during a single read, and when multiple reads can occur during a single write.

Sequencing is the ACM property that ensures that values are read out of the mechanism in the order in which they were written (although, of course, some values may be re-read, and others not read at all, before they are over-written). The first ACM to preserve sequencing in Lamport's hierarchy, is the L-atomic one.

In 1987 H.R. Simpson proposed an ACM implementation that used four "slots" to ensure that there was always a place into which data could be written, and there was always "recent" data that a reader could read, while ensuring that partially written (corrupted) data was never read, [Sim87]. This "4-Slot" mechanism used four shared bit control variables to coordinate the reader and writer so that they accessed the right slots. These shared bit variables are themselves normally assumed to be ACMs in their own right.

A recent technical report by John Rushby has used Simpson's 1987 4-Slot ACM as an example in a tutorial introduction to the SAL modelling language and model-checker, [Rus02]. Rushby shows that the 4-Slot mechanism does not maintain the ordering of values communicated through it with L-safe shared bit control variables. He also demonstrates, however, that the 4-Slot does preserve sequencing when the bit variables are H-atomic (that is, protected by mutual exclusion). He suggests that other algorithms which only require L-safe bit variables should be sought.

It is argued here that an L-safe register is an *inappropriate* model of a shared bit variable because of metastability. Furthermore, it is shown that, when more realistic models of shared bit variables are adopted, the 4-Slot mechanism preserves sequencing and ensures that reads always return fresh values.

The formal notation used to explore these issues is CSP, [Hoa85] and [Ros98]. CSP is chosen for two main reasons: firstly, it is a widely known process algebra, thus maximising the readership for the formal definitions developed; and secondly, it is supported by a powerful commercial model-checker, FDR2, [FSE96].

The rest of this paper is organised in the following way. Section 2 introduces the phenomenon of metastability and explains how circuits can be engineered to minimise its significance, and Section 3 provides a series of increasingly realistic CSP models of shared bit variables, taking into account metastability and how it is handled. This necessitates a brief discussion on how the algorithms which use such variables should be modelled. Section 4 introduces Simpson's 4-Slot mechanism, and Section 5 discusses the formalisation of it in CSP. Section 6 discusses the modelling of data-coherence, data-sequencing and freshness in CSP, and reports on the results established by model-checking the 4-Slot mechanism with various bit models against these properties. Finally, Section 7 draws general conclusions about the effects that failing to model metastability can have on the relevance of formal models of asynchronous mechanisms.

# 2 Metastability

In this section the problematic behaviour of metastability is briefly reviewed.

Metastability is a fundamental phenomenon of classical systems which have two or more stable states, and which respond to "connected" inputs – that is, to inputs which are either continuous in time, or continuous in value (or both)[3]. Basically, on the boundaries between the "regions" of a device's state-space that lead to these stable states there are states which are not stable, but which leave the eventual stable state of the system undetermined. The time taken to reach this state is undetermined too[4]. It is these states which are called "metastable". It has been shown that the probability that a system remains in a metastable state decreases exponentially with time, [Män88].

An important class of systems that can exhibit metastability are digital electronic circuits that synchronise asynchronous inputs, [CM73] and [HEC89]. The two binary states are the stable states, and the asynchronous

[3]Non-classical, that is, quantum, systems do not in principle have to exhibit metastability, but other quantum effects, and in particular, quantum tunnelling mean that in practice an unbounded amount of time might be necessary to determine the quantum state entered, [Män88].

[4]The presence of random noise perturbations means that statistically such systems will eventually move into one of the stable states; although, in principle, this may take an unbounded length of time to occur, as noise may move the system away from the boundary of the stable state, as well as towards it.

input can (by definition) occur arbitrarily close to the synchronising (latching) clock pulse, causing the device to read a changing input that will not have a clear binary value. The synchroniser or latch then enters a metastable state, where its latched value lingers indefinitely between the two stable digital states. It needs to be recognised that it is possible for a metastable value itself (while it is an invalid digital value) to induce metastability in a circuit that reads it.

It needs to be emphasised that metastability is a "fundamental" phenomenon in that it is unavoidable in circuits which have to handle asynchronous inputs, [Mar81] and [KC87], although there are various approaches that can be adopted practically to reduce the problem, [Män88]. One option is to use a detector to detect when the read value is metastable, and hold up the reading system's clock until the metastability has resolved, [Cha87][5]. This requires the ability to be able to stop the reading system's clock, and the system has to be able to cope with potentially arbitrary pauses in its operation.

A more common option is to accept that a read of an asynchronous signal may enter a metastable state, and hence to ensure that long enough is left after the value is latched (read) before it is used (read). A suitable duration can ensure that the probability that the metastability has failed to be resolved is reduced as low as is needed. This is a particularly practical solution when the reader is a processor executing software, as processor clock speeds are typically slow compared with the time it takes for metastability to resolve with a high probability. In such systems, it is not infeasible to engineer the circuits so that the expected mean-time between reading a metastable value is in the order of the age of the universe, thus making failures due to this phenomenon insignificant compared with other failures.

In certain technologies, when circuits are not engineered to reduce the problem of metastability, the effects of metastability can be more systemic than logical models of the circuit might suggest. For example, when a large number of logic gates in a CMOS circuit are exhibiting metastability, the power-rails effectively become shorted, and hence parts of the circuit that are not even "logically" connected to the metastable gates can be affected.

# 3 Models of Metastable Bit Variables

As mentioned above, an L-safe ACM has the property that a read which does not clash with a write will always get the value stored in the device, while a read that does clash may get any value of the valid type, [Lam86b]. One approach to implementing L-safe ACMs is therefore to ensure that the ACM can only represent valid values of the type. Lamport identifies one important class of L-safe ACM as bit variables that are used to communicate data of a binary valued type. This observation has led to a widespread assumption that shared bit variables are L-safe ACMs[6].

An L-safe ACM can be represented in CSP by the following process, where reads and writes are modelled by their start and end events (*sw*, *ew*, *sr*, and *er*, respectively), thus allowing the "true" concurrency of the reads and writes to be modelled in spite of CSP's interleaving semantics.

```
BIT1(val) = sw?x -> BIT1_w(val, x) [] sr -> BIT1_r(val)

BIT1_w(val, x) = ew -> BIT1(x) [] sr -> BIT1_wr(val, x)

BIT1_r(val) = sw?x -> BIT1_wr(val, x) [] er!val -> BIT1(val)

BIT1_wr(val, x) = ew -> BIT1_r_clashed(x) []
```

[5] Chapiro notes in his paper that this is not the same as causing the system to skip clock cycles until after the metastability has been resolved. This is because the detector's "end of metastability" signal would itself be an asynchronous input into the circuit which blocks the clock from reaching the reading system, and hence it itself could be the cause of further metastability. In Chapiro's solution, therefore, the clock when it resumes need not be "in phase" with what it was previously.

[6] For example, even within the literature on the 4-Slot, Clark *et. al.* and the current authors have previously blithely asserted that the 4-Slot's control bit variables are L-safe, [CXYD98] and [HP02].

```
                (er!b0 -> BIT1_w(val, x) |~| er!b1 -> BIT1_w(val, x))

BIT1_r_clashed(val) = sw?x -> BIT1_wr(val, x) []
                      (er!b0 -> BIT1(val) |~| er!b1 -> BIT1(val))
```

Note that a write introduces the new value at *start_write (sw)*, while the read does not return the value it acquires until *end_read (er)*. Also note that any clash between the reader and writer means that the value returned by the read is chosen by internal non-determinism. This is because, although the value read is returned by the *er* event, the time this value is determined (acquired) may be at any time during the read, and hence may be during the time that the reader was clashing with the write.

It should also be noted that during a write which changes the value of the bit from $b0$ to $b1$, a series of clashing reads may get the values $b0 - b1 - b0 - b0 - b1$, for example. This property could, perhaps, be justified in terms of the reader observing some kind of "flicker" during the write.

One important deficiency of this model of a bit variable from a hardware point of view, is that this flicker behaviour is exhibited even when the value in the bit is being overwritten with the same value. A simple modification of the above definition removes this problem:

```
BIT2(val) =
     sw?x -> (if x == val then BIT2_w_stable(val) else BIT2_w(val, x)) []
     sr -> BIT2_r(val)

BIT2_w(val, x) = ew -> BIT2(x) [] sr -> BIT2_wr(val, x)

BIT2_r(val) =
     sw?x -> (if x == val then BIT2_wr_stable(val) else BIT2_wr(val, x)) []
     er!val -> BIT2(val)

BIT2_wr(val, x) = ew -> BIT2_r_clashed(x) []
                  (er!b0 -> BIT2_w(val, x) |~| er!b1 -> BIT2_w(val, x))

BIT2_r_clashed(val) = sw?x -> BIT2_wr(val, x) []
                      (er!b0 -> BIT2(val) |~|  er!b1 -> BIT2(val))

BIT2_w_stable(val) = ew -> BIT2(val) [] sr -> BIT2_wr_stable(val)

BIT2_wr_stable(val) = ew -> BIT2_r(val) [] er!val -> BIT2_w_stable(val)
```

It should be noted that this model is more deterministic than the L-safe one, because reads that clash with writes which do not change the value of the bit return the value that the bit retains. In other words, BIT2 is an L-safe register (i.e. it refines BIT1), but it does not itself specify all the L-safe behaviours.

Clearly, both these models fail to capture metastability. The value returned is always well-defined, and there is no mechanism for allowing an arbitrary period of "dither" while the value read becomes determined. Furthermore, there is no way that metastable values can be propagated should the value read be used too soon.

Rather than attempting to construct a model of bit variables that reflects general metastability, the approach adopted here is to model bit variables that have been engineered to reduce the problem of metastability. We start by adapting the previous model to reflect Chapiro's solution, [Cha87], mentioned above, of using a metastability detector to delay the system clock until any metastability has been resolved. The CSP for such a model is:

```
BIT3(val) =
   sw?x -> (if x == val then BIT3_w_stable(val) else BIT3_w(val, x)) []
   sr -> BIT3_r(val)
```

```
BIT3_w(val, x) = ew -> BIT3(x) [] sr -> BIT3_wr(val, x)

BIT3_r(val) =
   sw?x -> (if x == val then BIT3_wr_stable(val) else BIT3_wr(val, x)) []
   er!val -> BIT3(val)

BIT3_wr(val, x) = ew -> BIT3_r_clashed(x) []
                  (er!b0 -> BIT3_w(val, x) |~| er!b1 -> BIT3_w(val, x) |~|
                   dither -> BIT3_wr(val, x))

BIT3_r_clashed(val) = sw?x -> BIT3_wr(val, x) []
                      (er!b0 -> BIT3(val) |~| er!b1 -> BIT3(val) |~|
                       dither -> BIT3_r_clashed(val))

BIT3_w_stable(val) = ew -> BIT3(val) [] sr -> BIT3_wr_stable(val)

BIT3_wr_stable(val) = ew -> BIT3_r(val) [] er!val -> BIT3_w_stable(val)
```

Here *dither* is an event which models the fact that the metastable value has not decayed to a true binary value yet. Clearly, it should be hidden from the processes in the environment of *BIT3*. One would expect algorithms which used bits with this model, would behave similarly to algorithms which used the *BIT2* model above, except that they would be able to diverge. This would reflect the fact that metastability can theoretically take an unbounded length of time to decay[7].

It needs to be emphasised that it is not strictly true to say that this is a model of an L-safe ACM, as it may fail to respond to a read – that is, it may *diverge*.

Chapiro's solution to metastability, however, is not widely adopted. One is therefore led to consider another model of bit variables, which takes into account metastability. The simplest way to do this within CSP is to extend the alphabet of the channels associated with the *sw* and *er* events so that they include an extra dithering value, *d*. The model of a bit therefore becomes:

```
BIT4(val) =
  sw?x -> (if x == val then BIT4_w_stable(val) else BIT4_w(val, x)) []
  sr -> BIT4_r(val)

BIT4_w(val, x) = ew -> BIT4(x) [] sr -> BIT4_wr(val, x)

BIT4_r(val) =
  sw?x -> (if x == val then BIT4_wr_stable(val) else BIT4_wr(val, x)) []
  er!val -> BIT4(val)

BIT4_wr(val, x) = ew -> BIT4_r_clashed(x) []
                  (er!b0 -> BIT4_w(val, x) |~| er!b1 -> BIT4_w(val, x) |~|
                   er!d -> BIT4_w(val, x))

BIT4_r_clashed(val) = sw?x -> BIT4_wr(val, x) []
                      (er!b0 -> BIT4(val) |~| er!b1 -> BIT4(val) |~|
                       er!d -> BIT4(val))

BIT4_w_stable(val) = ew -> BIT4(val) [] sr -> BIT4_wr_stable(val)

BIT4_wr_stable(val) = ew -> BIT4_r(val) [] er!val -> BIT4_w_stable(val)
```

At this point it is necessary to consider how the values read from bit variables are handled in the reading process. For the first three models, a simple process which reads and then uses a bit variable would be:

[7] This seems to be the model of metastability that has been captured in Petri-nets in [CXYD98].

```
READER1 = sr -> er?x -> use1(x) -> use2(x) -> READER1
```

However, the value returned by a bit variable may now be a metastable value, which could (but is not guaranteed to) decay into a standard binary value at some point in the future. A way to handle such behaviour systematically within a model is to use a process to model each "local bit variable" ("x" in the example above) used to store values read from shared bits.

In general, local bit variables may be set to a value, read, and may non-deterministically resolve metastable values into binary ones. Our first model of a local bit (LB) is therefore:

```
LB1(val) =  if val == d then
               (LB1(b0) |~| LB1(b1) |~| (set?x -> LB1(x) [] get!val -> LB1(val)))
            else
               (set?x -> LB1(x) [] get!val -> LB1(val))
```

Any reading processs would need to be changed to make use of such local variable processes. For example, READER1 above becomes READER2 below:

```
READER2 = sr -> er?x -> set!x -> get?x -> use1(x) -> get?x -> use2(x) -> READER2
```

Here the first two actions are reading the shared bit variable; this value is then "set" in a local bit variable, and this is then read back into the reading process (thus giving the opportunity for the potentially metastable value to resolve to a binary one). The value is then "used" as before. However, another "get" from the local variable precedes the next use of "x". Generally, every use of "x" in the original algorithm needs to be prefixed with such a "get", as this allows metastability resolution that occurs in the local variable process to influence the algorithm.

The problem with the above solution (*BIT4*, *LB1*, *READER2*) is that multiple reads of a metastable variable are treated as realistic behaviours which need to be taken into account. However, as was mentioned above, by leaving a suitable time between reading a shared bit and making use of the value obtained, the chances that it will still be metastable when it is used can be reduced to an arbitrarily low probability. This suggests that when such delays are engineered into the circuit, the following would be a more appropriate model for a local bit variable:

```
LB2(val) = set?x -> (if x == d then LB2(b0) |~| LB2(b1) else LB2(x)) []
           get!val -> LB2(val))
```

This process resolves metastable values as they are set, ensuring that all subsequent reads will get the same non-deterministically chosen value.

The final practical engineering issue relating to shared bit variables that needs to be incorporated into our model challenges the assumption that they are pure ACMs. In particular, it needs to be realised that digital electronic circuits have maximum speeds at which they may be operated, as the components within them will have minimum "set up and hold" times that have to be observed, if they are to be operated within their specifications, and they will have maximum switching or propagation times. Taken together these timing constraints limit the number of reads that may access a switching value. When the reads and writes arise from processor instructions, the relatively slow speed of processor clocks mean that at most one read will occur while a value is switching, and at most one write will occur while a value is being "read"[8]. It should be emphasised, therefore, that such shared bits are still ACMs in the sense that they do not force a synchronisation between the reader and writer accessing them. Furthermore, the ratio between the reader and writer frequencies can be arbitrary, providing that none of the frequencies excede the maximum operating frequencies of the circuitry. We therefore modify the *BIT4* model given above to reflect these constraints:

[8]This assumes that switching and latching are triggered by the edge of the clock, and hence the switching duration is independent of the clock frequency.

```
BIT5(val) =
   sw?x -> (if x == val then BIT5_w_stable(val) else BIT5_w(val, x)) []
   sr -> BIT5_r(val)

BIT5_w(val, x) = ew -> BIT5(x) [] sr -> BIT5_wr(val, x)

BIT5_r(val) =
   sw?x -> (if x == val then BIT5_wr_stable(val) else BIT5_wr(val, x)) []
   er!val -> BIT5(val)

BIT5_wr(val, x) = ew -> BIT5_r_clashed(x) []
                  (er!b0 -> BIT5_w_r_occurred(val, x) |~|
                   er!b1 -> BIT5_w_r_occurred(val, x) |~|
                   er!d -> BIT5_w_r_occurred(val, x))

BIT5_r_clashed(val) =
   er!b0 -> BIT5(val) |~| er!b1 -> BIT5(val) |~| er!d -> BIT5(val)

BIT5_w_stable(val) = ew -> BIT5(val) [] sr -> BIT5_wr_stable(val)

BIT5_wr_stable(val) = ew -> BIT5_r(val) [] er!val -> BIT5_w_stable(val)

BIT5_w_r_occurred(val, x) = ew -> BIT5(x)
```

Note for simplicity, only changing writes[9] are taken into account when ensuring multiple reads and writes do not overlap, as writes which do not change the value are already modelled in a way which means that they do not cause flicker or metastable behaviour (see BIT2).

We give one last model of a bit variable, not because it is particularly realistic, but because it reflects the assumption that many adopt when thinking about bits. Basically, it models a bit as an H-atomic variable.

```
BIT6(val) = sw?x -> ew -> BIT6(x) [] sr ->  er!val -> BIT6(val)
```

Note that for consistency with the other models, the reading and writing operations are still modelled by the events that mark their beginning and end, although this is not necessary in this case.

Readers might expect an L-atomic CSP model of a bit to be presented at this stage. Unfortunately, L-atomicity is not a finite state property, as it crucially depends upon being able to state that some linearization occurs of the potentially infinite sequence of writes that clash with a read. This sequence is even "potentially infinite" for variables that can only store two values, such as bits. A "pseudo-CSP" infinite state definition of an L-atomic bit is:

```
BIT7(vals) = sw?x -> BIT7_w(<x>^vals) [] sr -> BIT7_r(vals)

BIT7_w(vals) = ew -> BIT7(head(vals)) [] sr -> BIT7_wr(vals)

BIT7_r(vals) = sw?x -> BIT7_wr(<x>^vals) [] er!val -> BIT7(head(vals))

BIT7_wr(vals) = ew -> BIT1_r_clashed(vals) []
    (|~| x : { n | n <= length(vals) } @ er!vals(x) ->
                                        BIT7_w(vals[x..length(vals)]))

BIT7_r_clashed(vals) = sw?x -> BIT7_wr(<x>^vals) []
    (|~| x : { n | n <= length(vals) } @ er!vals(x) ->
                                        BIT7_w(vals[x..length(vals)]))
```

[9] That is, a write which changes the value stored in the bit variable.

# 4 Simpson's Four-Slot Mechanism

This paper focuses on Simpson's original "software" version of the 4-Slot ACM first presented in 1987, [Sim87], and subsequently re-described in a more substantial paper in 1990, [Sim90]. Simpson's 1990 paper also included a variant of the algorithms more suitable for implementation directly in hardware, as they were less susceptible to metastability, and exploited potential parallelism. A further variant of the 4-Slot mechanism has been described in [Sim97a]; this version integrates better into a wider class of communication protocols that Simpson has identified, [Sim03b]. These other versions of the 4-Slot are not considered here, however, as the effects of different models of metastability on the analysis of protocols are adequately demonstrated by the 1987 version of the 4-Slot mechanism.

Simpson has already provided a correctness analysis of the 1987 version of the 4-Slot, [Sim92], using a technique called "role model analysis", [Sim97b]. This contains an explicit discussion of metastability, and his analysis was predicated on the assumption that the implementation would be engineered so that metastability was not problematic. Other academic work analysing Simpson's 4-Slot mechanisms includes: [BJA96], [XC99], [Rus02], and [HP02].

The 4-Slot ACM uses four *slots*, and four shared bit control variables. The reader and writer algorithms use the control variables to ensure that the reader and writer never access the same slot at the same time, and hence the reader can never read values which are composed of partial items from more than one write. In other words, it achieves data-coherence through *co-ordination* rather than through synchronising *mutual exclusion*.

This means that the slots in a 4-Slot should never be subject to clashing reads, and hence that they do not need to be engineered to be L-safe. They only need the weaker property of *persistence*, [HP02]; that is, a non-clashing read must still get the last value written, but a clashing one may get any value – even an illegal (type invalid) one. Such "problematic" behaviour does not need to be engineered away as it should never occur.

The 4-Slot mechanism is shown in Table 1. It is deceptively simple: the write operation consists of only five actions, and read operation only four.

The mechanism is described as follows:

1. *the slots* are organised in two pairs of two slots (at least one of which is initialised);
2. the mechanism has four single bit control variables:

   ***reading*:** which indicates the pair the reader is reading (or last read) from.

   ***latest*:** which indicates the pair the writer is writing (or last wrote) to.

   ***writers_slots*:** a two element array of binary slot indices. The reader reads this array to choose the slot to read in the pair that it is currently accessing. The writer reads it to choose the slot to write to in the pair it us currently accessing. The writer also updates the array to indicate the slot it has written to. (It always chooses the other slot from which it last write in that pair.)

3. *the writer*:

   - chooses the pair and the slot within that pair to which it will write the new value - *Writer-Chooses-Pair* and *Writer-Chooses-Slot* in Table 1 (the write pre-sequence). It always chooses to write to the opposite pair to the one the reader last indicated it was reading from;[10]
   - writes the new item to the chosen slot - line labelled *Write* in Table 1; and

[10] This will be the pair the initial item was written to until the reader indicates the pair it is reading from for the first time.

Table 1: The 1987 4-Slot Mechanism

```
mechanism four_slot;
  type
    Pair_Index = (p0, p1);
    Slot_Index = (s0, s1);
  var
    slots :  array[Pair_Index, Slot_Index] of Data;
    writers_slots:  array[Pair_Index] of Slot_Index;
    latest, reading:  Pair_Index;

  procedure write (item:  Data);
  var
    pair_written :  Pair_Index;
    slot_written :  Slot_Index;
  begin
    pair_written := not reading;                        (Writer-Chooses-Pair)
    slot_written := not writers_slot[pair_written];     (Writer-Chooses-Slot)
    slots[pair_written, slot_written] := item;          (Write)
    writers_slots[pair_written] := slot_written;        (Writer-Indicates-Slot)
    latest := pair_written;                             (Writer-Indicates-Pair)
  end write;

  function read :  Data;
  var
    read_pair :  Pair_Index;
    read_slot:  Slot_Index;
  begin
    read_pair := latest;                                (Reader-Chooses-Pair)
    reading := read_pair;                               (Reader-Indicates-Pair)
    read_slot := writers_slots[read_pair];              (Reader-Chooses-Slot)
    return slots[read_pair, read_slot];                 (Read)
  end read;
end four_slot;
```

- indicates the slot and pair it has written the data to - *Writer-Indicates-Slot* and *Writer-Indicates-Pair* in Table 1 (the write post-sequence).

4. *the reader*:

   - chooses to read from the pair of slots last written to (or the pair the initial value was written to), indicates that it is reading from that pair, and then chooses to read from the slot in that pair that the writer last visited - *Reader-Chooses-Pair*, *Reader-Indicates-Pair* and *Reader-Chooses-Slot* in Table 1 (the read pre-sequence); and
   - reads the item from the chosen slot - line labelled *Read* in Table 1.

One potential source of metastability is in the *read* algorithm, where the local bit variable *read_pair* is assigned the value of the shared bit variable "latest" which may be changing (as the last statement of *write* algorithm sets it), thus causing the *read_pair* value to become metastable. However, *read_pair* is used by the very next instruction, meaning that the probability that it might still be metastable needs to be carefully considered. When the algorithm is implemented in software, the relatively slow speed of processors means that the time between instructions ensures that it is highly likely that metastability will have resolved in the intervening time.

# 5 A CSP Model of the 4-Slot ACM

In this section we present the main aspects of our model of the 1987 4-Slot in (machine readable) CSP. It builds upon Rod White's unpublished model of the 4-Slot in CSP, [Whi01]. We start by giving some of the basic constraints and types:

```
max_no_of_values = ...
data_values = {1..max_no_of_values}
datatype bit_values = b0 | b1 | d
datatype slot_index = s1 | s2 | s12
datatype pair_index = p1 | p2 | p12
```

Here *max_no_of_values* is a constant which defines the number of different data values which can be communicated through our model of the 4-Slot (the *data_values* type). This number needs to be kept low (2 or 3) for most of the tests. However, it has been increased to 10 for the sequencing test, where the number of data-items might be expected to be more crucial.

The *bit_values* type defines the values that our bits will be able to model – in particular, it contains *d*, that in some of the models is the value used to represent a "dithering" metastable signal.

The next two datatypes are "binary" valued enumeration types used to refer to slots and pairs, respectively – the s12 and p12 values represent a dithering index, and will be resolved non-deterministically by the model. These types are only used to add an aspect of type-checking to the model; they help keep track of when bit values are being used to model a reference to a slot or a pair in the mechanism.

We now present the datatypes, channel and process definitions needed for the four slots:

```
datatype slot_operations =
      sr_slot | er_slot.data_values | sw_slot.data_values | ew_slot

channel slots : pair_index.slot_index.slot_operations

channel clash_bang

Slot(pair_name, slot_name, val) =
   slots.pair_name.slot_name.sw_slot?x ->
      (slots.pair_name.slot_name.ew_slot -> Slot(pair_name, slot_name, x) []
       slots.pair_name.slot_name.sr_slot -> clash_bang -> STOP) []
   slots.pair_name.slot_name.sr_slot ->
      (slots.pair_name.slot_name.sw_slot?x -> clash_bang -> STOP []
       slots.pair_name.slot_name.er_slot!val -> Slot(pair_name, slot_name, val))

the_actual_slots = (Slot(p1, s1, 1) ||| Slot(p1, s2, 1) |||
                    Slot(p2, s1, 1) ||| Slot(p2, s2, 1))

channel start_write_slots, end_write_slots, start_read_slots, end_read_slots
```

Slots are accessed by read and write slot operations, which are modelled by their start and end events, *sr_slot*, *er_slot*, *sw_slot* and *ew_slot*, where the data value is communicated by the *er_slot* and *sw_slot* events. The 4-Slot algorithm communicates with the slots via the *slots* channel – this is parameterised by pair and slot indices, as well as the operation which is to be engaged in. The slots themselves are each modelled by the *Slot* process; this accepts reads and writes in any order, but when reads and writes overlap the process engages in the *clash_bang* event and then stops. This "clash_bang" event remains visible, and is used to check whether any of the various 4-Slot/bit models cause a loss of data-coherence through allowing the reader and writer to access the same slot at the same time. The slots themselves are modelled as four interleaving processes, each an instantiation of the Slot process, and each initialised with the value "1".

In the following models, however, the reader and writer algorithms access the slots via the guard processes, *read_slots* and *writer_slots*, rather than directly. These guard processes are responsible for arbitrarily resolving metastable references should such metastable values ever be used to index a slot (i.e. $p12$ or $s12$ into $p1$ or $p2$, and $s1$ or $s2$, respectively). These guard processes are not given here as they are not very interesting. The slots overall, however, are modelled in this way:

```
the_slots =
  (read_slots ||| write_slots) [| {| slots |} |] the_actual_slots \ {| slots |}
```

The datatype, channel and process definitions needed to define the shared and local bit variables are given next.

```
datatype shared_bit_operations = sr | er.bit_values | sw.bit_values | ew

channel reading, latest : shared_bit_operations
channel writers_slots : pair_index.shared_bit_operations

BITsX = (||| x : {reading, latest, writers_slots.p1,
                                    writers_slots.p2} @ BITX(x, b0))

datatype local_bit_operations = set.bit_values | get.bit_values

channel LB_write_pair, LB_write_slot,
        LB_read_pair, LB_read_slot : local_bit_operations

the_writers_local_bits1 =
       LB1(LB_write_pair, b0) ||| LB1(LB_write_slot, b0)

the_readers_local_bits1 =
      LB1(LB_read_pair, b0) ||| LB1(LB_read_slot, b0)


the_writers_local_bits2 =
       LB2(LB_write_pair, b0) ||| LB2(LB_write_slot, b0)

the_readers_local_bits2 =
      LB2(LB_read_pair, b0) ||| LB2(LB_read_slot, b0)
```

Note that a composite channel (*writers_slots*) models the shared array that stores the index of the last written slot in each pair. Also, note the bit and local bit process models are not repeated, as they have been given in Section 3. The only difference from the models given in Section 3 is that each of those models needs to be parameterised by the name of the bit, thus enabling multiple copies of the bits to be instantiated, as illustrated above (by the line which starts BITsX) In the complete model, the definition is actually repeated with X replaced with the numerals, 1 to 6.

With these definitions in place it is now possible to present the processes that model the 4-Slot reader and writer algorithms (which work with bits which do not return metastable values):

```
channel start_write, end_read : data_values
channel start_read, end_write

Fourslot_Writer =
    start_write?val -> reading.sr ->  reading.er?not_pair_written ->
    writers_slots.bp(toggle(not_pair_written)).sr ->
    writers_slots.bp(toggle(not_pair_written)).er?not_slot_written ->
    start_write_slots -> slot_written_pair!bp(toggle(not_pair_written)) ->
    slot_written_slot!bs(toggle(not_slot_written)) ->
    slot_written_val!val -> end_write_slots ->
    writers_slots.bp(toggle(not_pair_written)).sw!toggle(not_slot_written) ->
    writers_slots.bp(toggle(not_pair_written)).ew ->
    latest.sw!toggle(not_pair_written) -> latest.ew -> end_write ->
    Fourslot_Writer

Fourslot_Reader =
```

```
    start_read -> latest.sr -> latest.er?read_pair ->
    reading.sw!read_pair -> reading.ew ->
    writers_slots.bp(read_pair).sr ->
    writers_slots.bp(read_pair).er?read_slot ->
    start_read_slots -> read_slot_pair!bp(read_pair) ->
    read_slot_slot!bs(read_slot) -> read_slot_val?val ->
    end_read_slots -> end_read!val ->  Fourslot_Reader
```

The above models need to be modified to introduce local variables to handle metastable values being returned by reads of bits.

```
Fourslot_Writer_LB =
    start_write?val -> reading.sr -> reading.er?not_pair_written ->
    LB_write_pair.set!toggle(not_pair_written) ->
    LB_write_pair.get?pair_written -> writers_slots.bp(pair_written).sr ->
    writers_slots.bp(pair_written).er?not_slot_written ->
    LB_write_slot.set!toggle(not_slot_written) ->
    LB_write_slot.get?slot_written ->  LB_write_pair.get?pair_written ->
    start_write_slots -> slot_written_pair!bp(pair_written) ->
    slot_written_slot!bs(slot_written) -> slot_written_val!val ->
    end_write_slots -> LB_write_pair.get?pair_written ->
    LB_write_slot.get?slot_written ->
    writers_slots.bp(pair_written).sw!slot_written ->
    writers_slots.bp(pair_written).ew -> LB_write_pair.get?pair_written ->
    latest.sw!pair_written -> latest.ew -> end_write -> Fourslot_Writer_LB

Writer_LB1 =
      Fourslot_Writer_LB [| {| LB_write_pair, LB_write_slot |} |]
            the_writers_local_bits1 \ {| LB_write_pair, LB_write_slot |}

Writer_LB2 =
      Fourslot_Writer_LB [| {| LB_write_pair, LB_write_slot |} |]
            the_writers_local_bits2 \ {| LB_write_pair, LB_write_slot |}

Fourslot_Reader_LB =
    start_read -> latest.sr -> latest.er?read_pair ->
    LB_read_pair.set!read_pair -> LB_read_pair.get?read_pair ->
    reading.sw!read_pair -> reading.ew ->
    LB_read_pair.get?read_pair -> writers_slots.bp(read_pair).sr ->
    writers_slots.bp(read_pair).er?read_slot ->
    LB_read_slot.set!read_slot -> LB_read_slot.get?read_slot ->
    LB_read_pair.get?read_pair -> start_read_slots ->
    read_slot_pair!bp(read_pair) -> read_slot_slot!bs(read_slot) ->
    read_slot_val?val -> end_read_slots -> end_read!val -> Fourslot_Reader_LB

Reader_LB1 = Fourslot_Reader_LB [| {| LB_read_pair, LB_read_slot |} |]
                the_readers_local_bits1 \ {| LB_read_pair, LB_read_slot |}

Reader_LB2 = Fourslot_Reader_LB [| {| LB_read_pair, LB_read_slot |} |]
                the_readers_local_bits2 \ {| LB_read_pair, LB_read_slot |}
```

We are now at last in the position to present the overall 4-Slot model:

```
Four_Slot_BIT1 =
 (((Fourslot_Writer ||| Fourslot_Reader)
     [| {| read_slot_pair, read_slot_slot, read_slot_val, slot_written_pair,
     slot_written_slot, slot_written_val, writers_slots, reading, latest,
     start_write_slots, end_write_slots, start_read_slots, end_read_slots |} |]
```

```
 (the_slots ||| BITs1))  \ {| read_slot_pair, read_slot_slot, read_slot_val,
     slot_written_pair, slot_written_slot, slot_written_val, writers_slots,
     reading, latest, start_write_slots, end_write_slots, start_read_slots,
     end_read_slots |})
```

This definition needs to be repeated to combine the 4-slot algorithms with each shared and local bit model presented in Section 3.

# 6 Data-Coherence, Sequencing, and Freshness Results

This section describes CSP models of data-coherence, sequencing and freshness. It presents the results of using FDR2 to model-check the 4-Slot/bit models against these properties.

## 6.1 Data-Coherence

Data-coherence of the 4-Slot can be established by showing that it does not engage in the "clash_bang" event. This can be done by defining the *Incoherence_Spec* process below, and asserting that each 4-Slot model with its start and end read and write events hidden, is trace-refined by it. Model-checking will demonstrate that such an assertion fails for models which preserve data-coherence.

```
Incoherence_Spec = clash_bang -> STOP

assert (Four_Slot_BITx \ {| start_write, end_read, start_read, end_write |})
           [T=  Incoherence_Spec
```

The data-coherence results have also been confirmed using the more general definition of data-coherence in terms of the following semi-regular process (See Section 1).

```
SemiRegACM(vals) = start_write?x -> SemiRegACM_w(union({x}, vals)) []
                   start_read -> SemiRegACM_r(vals)

SemiRegACM_w(vals) = end_write -> SemiRegACM(vals) []
                     start_read -> SemiRegACM_wr(vals)

SemiRegACM_r(vals) = start_write?x -> SemiRegACM_wr(union({x}, vals)) []
                     ([] z : vals @ end_read!z -> SemiRegACM(vals))

SemiRegACM_wr(vals) = end_write -> SemiRegACM_r(vals) []
                      ([] z : vals @ end_read!z -> SemiRegACM_w(vals))
```

## 6.2 Sequencing

The obvious way to check whether a mechanism preserves the correct sequencing of data is to add a reader and writer process, where (a) the writer process writes a monotonically increasing sequence of values; (b) the reader checks each value that it reads; and (c) the reader stops with an "order_bang" if it ever reads a value older (smaller) than the value it read previously:

```
Write_Act(n) = start_write!n -> if n == max_no_of_values then STOP
                                else end_write -> Write_Act(n+1)

Read_Act(old_x) = start_read -> end_read?x ->
                  if x < old_x then order_bang -> STOP else Read_Act(x)

assert STOP [T= ((Write_Act(1) ||| Read_Act(0))
```

```
    [| {| start_write, end_read, start_read, end_write |} |]
Four_Slot_BITx)
       \ {| start_write, end_read, start_read, end_write |}
```

## 6.3 Freshness

Freshness is the desirable property that the reader should not return older (staler) data when more recently written data is available. However, as mentioned in the introduction, this description is ambiguous as it is not obvious what "available" means when multiple writes may overlap a single read, and *vice versa*.

Brooke, Jacobs, and Armstrong in [BJA96] explore four attempts to describe freshness in terms of the beginning and end events of reads and writes. Their definitions exploit the options that arise from considering the start or end of the write as marking the event when the data becomes available, and the start or the end of read as the point when the "availability" is assessed. They conclude that none are entirely satisfactory, [BJA96]. The 4-Slot can always fail to return the "freshest value" whichever of their definitions is adopted. However, they argue that this is not proof of a deficiency in the 4-Slot, because examination of the traces which fail (in particular, when the *ew* and *er* are chosen as the critical events) reveals that these "failures" are times when the 4-Slot gets data which is *too fresh*, according to the definition – i.e. the read returns data which has been written into the mechanism, but which has not been fully released. It seems reasonable to follow Brooke et. al. and conclude that freshness has not been defined adequately, rather than to conclude that the 4-Slot has failed to exhibit the desired property.

More recently, Rushby, [Rus02], has examined the 4-Slot against an attempted formalisation of freshness in terms of the beginning and end events of read and write actions. The *start_write* event in his formalisation stores the previous and new (about to be written) value of the ACM, and when the start and end read events occur they take note of these values. Rushby defines a read that returns one of these four values as being "fresh". Unfortunately this definition has an undesirable property; a read that gets a value from a clashing write which is not one of these values is not "fresh". This is clearly not correct, because if, for example, a read clashes with 4 writes, the first and the last two are deemed fresh but the second is not.

Freshness can be defined using outer read and write events in two ways: namely using the L-regular or L-atomic properties. In the first case, a read may return the value written by any clashing write (or the value in the ACM prior to the read starting.) This interprets "available" in terms that are *local* to each read operation. With the L-atomic definition, however, data is no longer considered to be fresh when more recently written data has been returned by a previous read. This clearly involves considerations that are *global* to a series of reads. It has been argued in [HP02] that it is the global L-atomic property which best captures the informal concept of "freshness". However, an advantage of adopting the L-regular (local) definition of freshness is that the *sequencing* property is an independent requirement, and is not subsumed within the freshness property.

Simpson, has recently pointed out, [Sim03a], that defining freshness purely in terms of these outer "beginning" and "end" events is not ideal. He notes that in a wide class of ACM implementations, the read and write actions consist of two phases – the "put" and "release" and the "acquire" and "get" phases, respectively. In the "put" phase, data is placed into the mechanism, and in the "release" phase, the writer informs the reader where the data has been put. In the "acquire" phase, the reader decides where to read the data from, and in the "get" phase it actually reads the data. Simpson notes that the "put" and "get" phases are irrelevant for the purposes of defining freshness, as the new value cannot possibly be obtained by the reader during the "put" phase, and the value obtained cannot be changed during the "get" phase. He therefore concludes that freshness should be defined in terms of the release and acquire phases, rather than the whole read and write actions. With this more nuanced model, Simpson prefers an L-regular (local) definition of freshness. However, even within this more nuanced model, the argument is still strong that the L-atomic (global) definition better captures the informal concept of freshness.

Clearly, there is ambiguity as to what the informal word "fresh" means in this context. Claims concerning "freshness" therefore always need to be clarified.

Fortunately, for finite datatypes, L-regular is a finite state property. It is therefore possible to define a CSP process which captures the L-regular property, thus making (local) freshness something that can be determined by model-checking. A CSP process which captures the "L-regular" property is given below – it builds up a set of values which the reader may acquire, as more and more writes clash with a read:

```
RegACM(val) =  start_write?x -> RegACM_w(union({x}, {val}), x) []
               start_read -> RegACM_r(val)

RegACM_w(vals, x) =  end_write -> RegACM(x) []
                     start_read -> RegACM_wr(vals, x)

RegACM_r(val) =  start_write?x -> RegACM_wr(union({x}, {val}), x) []
                 end_read!val -> RegACM(val)

RegACM_wr(vals, x) = end_write -> RegACM_r_clashed(vals, x) []
                     ([] z : vals @ end_read!z -> RegACM_w(vals, x))

RegACM_r_clashed(vals, x) = start_write?z -> RegACM_wr(union({z}, vals), z) []
                            ([] z : vals @ end_read!z -> RegACM(x))

RegACM_Spec = RegACM(1)
```

A mechanism which is L-regular and which preserves the sequencing of data passed through it is L-atomic. Therefore, both the local and global definitions of freshness can be determined by model-checking[11].

An alternative way of modelling (global) freshness is to check whether the mechanism behaves like an asynchronous reader and writer whose core state is modelled by an H-atomic variable, as in the following definitions:

```
H_Atomic_Var(var_name, val) =
     var_name.wr_op?x -> H_Atomic_Var(var_name, x) []
     var_name.rd_op!val -> H_Atomic_Var(var_name, val)

Read = start_read -> pool.rd_op?val -> end_read!val -> Read

Write = start_write?val -> pool.wr_op!val -> end_write -> Write

H_Atomic_state = H_Atomic_Var(pool, 1)

H_Atomic_spec = (((Read ||| Write) [| {| pool |} |] H_Atomic_State) \ {| pool |})
```

We check our models against this definition in the next subsection.

## 6.4 Results and Analysis

This subsection summarises the results of model-checking the 1987 4-Slot mechanism, with each of the bit models, against the properties described above. Table 2 summarises the properties of each model, and Table 3 describes the results.

In the results table, the L-atomic column is obtained by a conjunction of the results of the L-regular and Sequencing columns. The properties being reported are the properties of the 4-Slot, with the shared control bits modelled in the way indicated in each row.

[11] This does not contradict the observation made above that an L-atomic process is necessarily infinite state. A process which can exhibit *all* the behaviours of an L-atomic ACM is infinite state; but determining whether all the behaviours of a finite state process are valid behaviours of an L-atomic ACM is a finite check.

Table 2: The Descriptions of the Different Bit Models

| Model | Model Description |
|---|---|
| BIT1 | L-safe. Allows arbitrary clashes. No model of metastability. |
| BIT2 | As BIT1, except remains stable when over-written with the same value |
| BIT3 | As BIT2, except metastability causes arbitrary clock stretching |
| BIT4 LB1 | As BIT2, except has metastable values, which may be re-read |
| BIT4 LB2 | As BIT4 LB1, except metastable values cannot be re-read |
| BIT5 LB1 | As BIT4 LB1, except timing constraints prevent multiple clashes |
| BIT5 LB2 | As BIT4 LB2, except timing constraints prevent multiple clashes |
| BIT6 | H-atomic, causing mutually exclusive access to each bit |

Table 3: 4-Slot Coherence, Sequencing and Freshness Results

| 1987 4-Slot with these bit models: | Coherence [T= | L-Regular [T= | Sequencing [T= | L-Atomic [T= | H-Atomic [T= |
|---|---|---|---|---|---|
| BIT1 | √ | × | × | × | × |
| BIT2 | √ | √ | × | × | × |
| BIT3 | √ | √ | × | × | × |
| BIT4 LB1 | × | × | × | × | × |
| BIT4 LB2 | √ | √ | × | × | × |
| BIT5 LB1 | × | × | × | × | × |
| BIT5 LB2 | √ | √ | √ | √ | √ |
| BIT6 | √ | √ | √ | √ | √ |

In summary, this work shows that Simpson's original 1987 4-Slot preserves data-coherence, sequencing, and freshness (whether defined locally, as the L-regular property, or globally, as the L-atomic property) when modelled with the most realistic model of shared bit variables. It also shows that this mechanism even maintains data-coherence when shared bit control variables are only assumed to be L-safe; only when metastable values may be re-read can the mechanism violate data-coherence.

These results illustrate how important it is to model metastability adequately when formally analysing the behaviour of ACMs or wait-free protocols. In particular, note that the 4-Slot would fail to preserve data-coherence should metastability be present and the device not be engineered to prevent metastable values being re-read before the metastability is resolved. This means that even formal models which assume pessimistic L-safe behaviour of bits, but which fail to take these issues into account, may conclude that a mechanism has certain desirable properties (such as data-coherence) when it does not. Furthermore, the fact that the 4-Slot is neither L-regular nor L-atomic with L-safe bit models, but that it is both with the BIT5 LB2 model of bits, indicates that failure to model metastability and realistic engineering constraints may lead to calls (such as Rushby's, [Rus02]) to improve upon perfectly adequate algorithms.

We have not looked at the formal analyses of other protocols, but clearly, this work shows that there might be a problem with all work which fails to take metastability into account.

# 7 Conclusions

When constructing formal models of asynchronous communication mechanisms it is not common to take metastability into account. However, it has been shown that the metastable behaviour of shared bit variables found in such mechanisms can undermine the correctness of the protocol in question, and can also undermine the correctness of negative analysis results. It has been shown, however, through a series of increasingly realistic models of bits in CSP, that metastability, and the operational constraints that need to be adopted to cope with it, can be handled formally. Furthermore, these models have been applied to investigate the impact of metastability on Simpson's 4-Slot mechanism.

This paper, as well as raising important questions about the modelling of ACMs and wait-free protocols using L-safe registers has also produced an independent formal analysis of Simpson's 4-Slot. This analysis confirms, using standard model-checking technology, that the protocol maintains data-coherence, sequencing, and data-freshness. It confirms the results, [Sim92], of Simpson's role model analysis, [Sim97b].

The paper has also made modest contributions to the specification of the data-coherence and freshness properties for ACMs, by suggesting (a) that the *semi-regular* property is an abstract specification of *data-coherence*, and (b) that *L-regular* and *L-atomic* specifications are appropriate ways of formalising *freshness*.

# 8 Acknowledgements

MBDA UK Ltd. and the Bae Systems DCSC funded this research. Our ideas have benefited from conversations with Profs H.R. Simpson and C.B. Jones, Drs. F. Xia and I. Clark, and Mssrs. Eric Campbell and Rod White. We also acknowledge our debt to Prof. John Rushby's work on the 4-Slot in SAL.

# References

[BJA96] Phillip Brooke, Jeremy L. Jacob, and James M. Armstrong. Analysis of the Four-Slot Mechanism. In *Proceedings of the BCS-FACS Northern Formal Methods Workshop*, 1996.

[Cha87] Daniel M. Chapiro. Reliable High-Speed Arbitration and Synchronization. *IEEE Transactions on Computers*, 36(10):1251–1255, October 1987.

[CM73] Thomas J. Chaney and Charles E. Molnar. Anomalous Behavior of Synchronizer and Arbitor Circuits. *IEEE Transactions on Computers*, 22(4):421–422, April 1973.

[CXYD98] I.G. Clark, F. Xia, A.V. Yakovlev, and A.C. Davis. Petri Net Models of Latch Metastability. *Electronic Letters*, 34(7):635–636, 1998.

[FSE96] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: The FDR 2.0 User Manual*, August 1996.

[HEC89] Jens U. Horstmann, Hans W. Eichek, and Robert L. Coates. Metastability Behaviour of CMOS ASIC Flip-Flops in Theory and Test. *IEEE Journal of Solid-State Circuits*, 24(1):146–157, February 1989.

[Hoa74] C.A.R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, 1974.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science, 1985.

[HP02] N. Henderson and S.E. Paynter. The Formal Classification and Verification of Simpson's 4-Slot Asynchronous Communication Mechanism. In L.-H. Eriksson and P.A. Lindsay, editors, *Proceedings of FME'02*, number 2391 in Lecture Notes in Computer Science, pages 350–369. Springer, 2002.

[JHJ89] Mark B. Josephs, C.A.R. Hoare, and He Jifeng. A Theory of Asynchronous Processes. Technical Report PRG-TR-6-89, Programming Research Group, Oxford University Computing Laboratory, February 1989.

[KC87] Lindsay Kleeman and Antonio Cantoni. On the Unavoidability of Metastable Behaviour in Digital Systems. *IEEE Transactions on Computers*, 36(1):109–112, January 1987.

[Lam86a] L. Lamport. On Interprocess Communication - Part 1: Basic Formalism. *Distributed Computing*, 1:77–85, 1986.

[Lam86b] L. Lamport. On Interprocess Communication - Part 2: Algorithms. *Distributed Computing*, 1:86–101, 1986.

[Män88] Reinhard Männer. Metastable States in Asynchronous Digital Systems: Avoidable or Unavoidable? *Microelectronic Reliability*, 28(2):295–307, 1988.

[Mar81] Leonard R. Marino. General Theory of Metastable Operation. *IEEE Transactions on Computers*, 30(2):107–115, February 1981.

[Ros98] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall Series in Computer Science, 1998.

[Rus02] John Rushby. Model-Checking Simpson's Four-Slot Fully Asychronous Communication Mechanism. Technical Report Issued, Computer Science Laboratory – SRI International, July 2002.

[Sim87] H.R. Simpson. Fully Asynchronous Communication. In *Proceedings of the IEE Colloquium on MASCOT in Real-Time Systems*, May 1987.

[Sim90] H.R. Simpson. Four-Slot Fully Asynchronous Communication Mechanism. *IEE Proceedings*, 137 Part E(1):17–30, January 1990.

[Sim92] H.R. Simpson. Correctness Analysis for Class of Asynchronous Communication Mechanism. *IEE Proceedings*, 139 Part E(1):35–49, January 1992.

[Sim97a] H.R. Simpson. New Algorithms for Asynchronous Communication. *IEE Proceedings of Computer Digital Technology*, 144(4):227–231, July 1997.

[Sim97b] H.R. Simpson. Role Model Analysis of an Asynchronous Communication Mechanism. *IEE Proceedings of Computer Digital Technology*, 144(4):232–240, July 1997.

[Sim03a] H.R. Simpson. Freshness Specification for a Class of Asynchronous Communication Mechanisms. *To be submitted to IEE Proceedings of Computer Digital Technology*, 2003.

[Sim03b] H.R. Simpson. Protocols for Process Interaction. To Appear in IEE Proceedings on Computers and Digital Techniques, 2003.

[Whi01] R.G. White. ASRAAM: Software Requirements and Design, Specification and Test Strategy for the EPU Infrastructure Software (New EPU). Technical Report DL 21025, Matra BAe Dynamics, 2001.

[XC99] F. Xia and I. Clark. Complementing the Role Model Method With Petri-Net Techniques in Studying Issues of Data Freshness of the Four-Slot Mechanism. Technical Report CS-TR-654, Department of Computer Science – University of Newcastle, January 1999.