School of Computing Science,
University of Newcastle upon Tyne

Metastability in Asynchronous Wait-Free Protocols

S.E. Paynter, Neil Henderson and J.M. Armstrong

Technical Report Series

CS-TR-859

September 2004

Metastability in Asynchronous Wait-Free Protocols

S.E. Paynter[1] and N. Henderson[2] and J.M. Armstrong[3]
[1] MBDA UK Ltd, Filton, Bristol, UK. *stephen.paynter@mbda.co.uk*
[2] Bae Systems DCSC*, University of Newcastle, UK.
[3] Centre for Software Reliability, University of Newcastle, UK.
{*neil.henderson, J.M.Armstrong*}*@ncl.ac.uk*

today

Abstract

Metastability can undermine the correctness of protocols which are demonstrably correct when metastability is ignored, e.g. when shared bits are assumed to be L(amport)-safe registers. We establish this using the CSP process algebra and the FDR2 model-checker, which we use to investigate the impact of various models of shared bits on different wait-free protocols, including Lamport's regular register, Simpson's 4-slot ACM; Kirousis et. al's ACM; Tromp's Atomic Bit and 4-Track ACMs; and Haldar and Subramanian's ACM. On the one hand, these ACMs exhibit different failure modes when metastability effects defeat hardware design measures to contain them. In this case the commonly used L-safe bit abstraction might rule out possible failure behaviours. On the other hand, most of these ACMs function correctly when metastability is resolved between instructions. In this case the L-safe abstraction permits failure behaviours which would not actually occur. Consequently, impossibility results concerning ACMs which are based on L-safe bit models may be pessimistic. We demonstrate this by showing that Simpson's 4-slot ACM functions correctly when constraints associated with metastability containment hold.

Keywords: protocol verification, model checking, formal models, asynchronous operation

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. Sometimes a protocol is constructed out of simple (binary) asynchronous signals to establish higher-level synchronous communication. On other occasions, high-level asynchronous protocols are deployed. Protocols which support asynchronous communication are variously called *wait-free protocols* e.g. [Pet83], *asynchronous registers* e.g. [Lam86b], or *asynchronous communication mechanisms* (ACMs) e.g. [Sim90].

Conceptually, ACMs support the communication of data between writing and reading processes in a way which prevents any temporal interaction between them. The progress of the write algorithm is unaffected by the reader accessing the ACM at the same time, and vice versa. The relative rates of the writer and reader are unconstrained. Not only may reads and writes overlap, but multiple consecutive writes may overlap a read and multiple consecutive reads may overlap a write[1]. ACMs are essentially *shared-variables*, and hence have the properties that:

*Dependable Computing Systems Centre

[1] ACMs therefore are *Lock-Free*, which only requires that reader and writer accesses do not force each other to synchronise through locking any variables. ACMs also satisfy the stronger property that rules out *any* temporal

- Each write puts one (possibly aggregate) value into the ACM;
- Each read gets one (possibly aggregate) value from the ACM;
- A value written into an ACM may be read many times; and
- Writing a value conceptually destroys (makes unavailable for reading) values written previously[2].

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, which use mutual exclusion to ensure that inconsistent data cannot be read, [Hoa74].

ACMs are of particular interest because they support the integration of sub-systems or processes which run at different frequencies or which are sporadic, while decoupling the temporal interactions between them. They therefore provide a means of building systems which are robust against deadlock due to the failure of one of the communicating systems.

Unfortunately, it is impossible to design a classical (non-quantum) system that reads an asynchronous message which may not suffer from a phenomenon called "metastability", [Män88][3]. Metastability can have various detrimental effects on the performance or behaviour of the system. However, although this is well known by electronic engineers who design circuits which engage in asynchronous communication, it is less well known by those who design and analyse wait-free protocols[4]. Much of the formal analysis of ACMs is done on the assumption that shared bits in such protocols behave as L-safe registers, [Lam86b][5]. However, an L-safe register does not model metastability.

The authors have shown in a previous paper, [PHA04], that a protocol which can be demonstrated to have desirable properties when its shared bits are assumed to be L-safe registers, may fail to have these properties when metastability is not contained; Simpson's 1987 4-Slot ACM, [Sim87] and [Sim90], preserves data-coherence with L-safe bits, but does not do so when metastable variables can be re-read two or more times before metastability resolves. We suggested in that paper that there might be a problem with any wait-free protocol whose behaviour had not been checked when metastability is not contained, even those with correctness proofs, if those proofs were based on the L-safe model of bits. In this paper we take up the challenge, and apply the same metastability analysis techniques that we applied to Simpson's 4-slot ACM to other well-known wait-free protocols. The new protocols we analyse are: Lamport's regular ACM, [Lam86b], Kirousis et. al's ACM, [KKV87]; Tromp's Atomic Bit and 4-Track ACMs, [Tro89]; and Haldar and Subramanian's ACM, [HS94].

The rest of this paper is organised in the following way. Section 2 introduces the phenomenon of metastability and explains how circuits are engineered to minimise its impact. Section 3 introduces the terminology that is used to characterise and compare wait-free protocols. Section 4 describes a series of increasingly realistic CSP models of shared bit variables.

interference. To appreciate the distinction, consider the lock-free protocol which has a reader which can detect when it has clashed with the writer, and keeps on re-reading until a non-clashing read occurs. Such an algorithm is clearly not wait-free. Algorithms which place an upper-bound on any temporal interference are also usually called "wait-free".

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

[3]However, while quantum systems do not in principle have to exhibit metastability, there seems to be other quantum effects, such quantum tunnelling, that mean that in practice an unbounded amount of time might be necessary to determine the quantum state entered.

[4]Even when there is some awareness of metastability it is sometimes ignored. For example, Kirousis et. al., [KKV87], write "We do not consider this level of physical detail here, but consider idealised bistables."

[5]L-safe registers are introduced below (Section 3).

These show how the effects of metastability can be taken into account in a model of an ACM. This necessitates a brief discussion on how the algorithms which use such variables should be modelled. Section 5 gives the specification models for semi-regular (data-coherent), L-regular, and L-atomic (global freshness and sequencing) registers. Section 6 reports on the results established by model-checking the different ACMs with the various bit models. Finally, some conclusions on modelling bits in ACMs are drawn in Section 7.

2 Metastability

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). Such systems have: (a) stable states; (b) regions of unstable states which must lead to a stable state; and (c) metastable states between the unstable regions. The unstable region (and hence stable state) which will be entered from a metastable state, and the length of time it takes to enter such a region is undetermined. However, it has been shown that the probability that a system remains in a metastable state decreases exponentially with time, [Män88] and [KBY02].

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 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 may linger 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. Note that metastability is not induced in the input (i.e. writing) circuit.

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 practical approaches that can be adopted to reduce the problem, [Män88]. One option is to use a detector to detect when the value read is metastable, and hold up the reading system's clock until the metastability has resolved, [Cha87][6]. This requires the ability to stop the reading system's clock, and the system has to be able to cope with arbitrary long pauses in its operation.

A more common option is to accept that a read of an asynchronous signal may enter a metastable state, and ensure that after the value is latched (i.e. read) a period long enough for metastability to resolve elapses before the value is used as an input to another device. A suitable duration can ensure that the probability that the metastability has failed to be resolved is reduced as low as is needed[7].

In the past, this approach has not entailed adding extra delays into the reader, when the reader is implemented in software. However, increases in processor clock frequencies mean that careful consideration now needs to be given to how asynchronous signals are handled in software implementations[8]. This strongly suggests that metastability can no longer be safely ignored

[6]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 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.

[7]For example, it is not impractical to design circuits which have a mean time between failures due to metastability of 10^{204} or 10^{420} years, [Gin03]. (The age of the universe is thought to be in the order of 10^{10} years.)

[8]Kinniment et. al., [KBY02], suggest that for processors with 1GHz clocks the time for metastability to resolve may need to be over 30 times the response time of the circuit to give a mean time between failures of less than

when analysing wait-free protocols implemented in software.

In certain technologies, when circuits are not engineered to contain the effects of metastability, these effects 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, significant current may be drawn. Furthermore, the delay in leaving a metastable state may upset timing assumptions through the logic. In these ways remote parts of the circuit can be affected.

3 Specifying and Classifying ACMs

This section introduces two different approaches to classifying ACMs. The terminology introduced is used to describe the ACMs introduced below.

All the ACMs considered in this paper are wait-free, and only support a single reader and a single writer.

3.1 A Behavioural Classification Scheme

The behavioural classification scheme we use consists of a cumulative hierarchy of five properties: *persistent*, *L-safe*, *semi-regular*, *L-regular*, and *L-atomic*, [HP02]. The *L-safe*, *L-regular*, and *L-atomic* properties where introduced by Leslie Lamport in two seminal papers in 1986, [Lam86a, Lam86b], where they were just called *safe*, *regular*, and *atomic*. We adopt the "L-" prefix (for Lamport) to honour this contribution, and to ease the systematic distinction between the Lamport and Hoare atomicity.

A *persistent* ACM is one which behaves like a normal local variable when no read occurs contemporaneously with a write, but when a read clashes (overlaps in time) with a write it may return any value which the ACM can represent. This value need not be one of the values of the type that the writer is attempting to communicate through the ACM.

An *L-safe* ACM is a persistent ACM, except that a clashing read may only return an arbitrary value of the type that is being communicated. When the ACM can only represent values of this type, a persistent ACM is automatically an L-safe one. Shared single bits used to communicate binary values have been assumed to be L-safe ACMs in much of the literature since [Lam86b].

A *semi-regular* ACM is an L-safe ACM, except that a clashing read may only return an arbitrary value that has been previously written into the ACM[9] (or the initial one). We have noted elsewhere that an ACM which does not corrupt data communicated through it (i.e. which preserves data-coherence) is semi-regular, [PHA04].

An *L-regular* ACM is a semi-regular 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) a value written by one of the (possibly multiple) writes that clash with that read. This can be considered to be a good definition of the *local freshness* property, which requires that each read gets the most recently "available" value, where "availability" is defined relative to each read, [PHA04]. Values may be read out of an L-regular ACM in a different order than they were written into it. This is because a later read may get a value which was written earlier than a value which an earlier read has already returned.

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

1 in 2 years.

[9] That is, by a write that starts before the read finishes.

arising from mutual exclusion. An L-atomic ACM preserves the ordering or sequencing of data communicated through it. This can be considered to be a good definition of the *global freshness* property, which requires that each read gets the most recently "available" value, where "availablility" is defined relative to the sequence of reads, [PHA04]. In other words, the global freshness definition does justice to the intuition that one cannot claim that a value is not available for reading if an earlier read has already returned it.

3.2 An Implementation Classification Scheme

The classification scheme we review here is widely used in the literature on wait-free protocols, for example [HV95].

ACMs which are used to communicate more than two values are usually called *multivalued*, while ACMs which only communicate two values are normally called *bits*.

Those implementations which use different variables to pass data and to co-ordinate access to the data, are called *buffer-based*. The variables used to hold the data being communicated are called *buffers* in this scheme, although they are also known as "slots" or "tracks" in particular algorithms.

In a buffer-based ACM no co-ordination information is passed through the buffers, and no data is passed through the control variables

A buffer-based shared variable where the reader and writer never access the same buffer (not control-variable) at the same time is called either *pure*, [BP89], or *conflict-free*, [HS94]. One advantage of an ACM which is conflict-free is that its buffers need only be implemented by persistent ACMs, as clashing accesses on a buffer will not occur (by definition).

Furthermore, a buffer-based shared variable where the write or read algorithm only writes or reads one buffer per execution may be called *write-once* or *read-once*, respectively. A non conflict-free protocol is unlikely to be able to be read-once, as some re-reading will be necessary should a conflict have occurred, since the conflict might have resulted in the read obtaining inconsistent data. The write-once read-once properties are highly desirable in multivalued ACMs which are required to communicate large complex data-values.

3.3 Some ACM Impossibility Results

In 1983 Peterson showed that buffer-based, 1-Writer ACMs needed to have $n+2$ buffers, where n is the the number of readers, [Pet83]. In 1989 Burns and Peterson showed that when the conflict-free constraint was added an ACM had to have at least $2n+2$ buffers. All multivalued 1-Reader ACMs considered in this paper use four buffers, and are therefore *buffer-optimal*.

In [HV01], Haldar and Vidyasankar have shown that it is impossible to realise a conflict-free write-once L-atomic variable from only four buffers and four L-safe shared control bits[10].

4 Modelling Shared Bit Variables

In this section we present nine models of shared bits in CSP_m, the machine readable ASCII version of the CSP language, [Ros98]. These are then used in the models of the ACMs that are analysed in this paper. The first model considers a shared bit to be a simple H-atomic variable. The next four differ over whether they are stable or flicker when overwritten with the value they already hold, and over whether they allow multiple or only single clashes. The last four repeat these four options, but add metastability. There is also a discussion of how algorithms should be modelled that use bits which may return metastable values.

[10] It is a pity that such an important proof is unpublished.

4.1 BIT_{H_Atomic}: An H-Atomic Bit Model

We first consider an H-atomic variable as a model of a bit. This is not realistic for most implementations of shared bits, but it reflects the assumption that many adopt when thinking about bits, and is included here for completeness.

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

Note, we assume "val" takes one of the binary values, $b0$ or $b1$. Also note, that for consistency with the other bit models in the section, the writing and reading operations are modelled by the events that mark their beginning and end, (sw, ew, sr, and er), respectively. In the other models this allows the "true" concurrency of the reads and writes to be modelled in spite of CSP's interleaving semantics, although, of course, this is not necessary in this case.

4.2 BIT_{fm}: Flickering, Multiple-Clash (L-Safe) Bit Model

The BIT_{fm} bit model "flickers" when the bit is overwritten with a new value, even if the new value is the same as it is already holding. This flickering behaviour is reflected in the "f" in its name. It also allows multiple reads to clash with a single write, and multiple writes to clash with a single read; this multiple clashing behaviour is represented by the "m" in its name.

These properties mean, for example, that a series of reads that clash with a single write might get the values $b0 - b1 - b0 - b0 - b1$.

BIT_{fm} is an L-safe ACM. Non-clashing reads always get the value stored in the bit, while a read that does clash may get any value of the valid type, [Lam86b]. 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[11].

BIT_{fm} is modelled by the following process:

```
BITfm(val) = sw?x -> BITfm_w(val, x) []
             sr -> BITfm_r(val)

BITfm_w(val, x) = ew -> BITfm(x) []
                  sr -> BITfm_wr(val, x)

BITfm_r(val) = sw?x -> BITfm_wr(val, x) []
               er!val -> BITfm(val)

BITfm_wr(val, x) = ew -> BITfm_r_clashed(x) []
                   (er!b0 -> BITfm_w(val, x) |~|
                    er!b1 -> BITfm_w(val, x))

BITfm_r_clashed(val) = sw?x -> BITfm_wr(val, x) []
                       (er!b0 -> BITfm(val) |~|
                        er!b1 -> BITfm(val))
```

The process starts by accepting either a *start_write (sw)* or a *start_read (sr)* event. Depending upon which is chosen it moves to BIT_{fm}_w or BIT_{fm}_r, which offer the appropriate events taking into account which action has already started. These then progress to subsequent states. 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

[11] For example, even within the literature on the 4-Slot, Clark *et. al.* and the current authors have previously asserted or argued that the 4-Slot's control bit variables are L-safe, [CXYD98], [Cla00], and [HP02].

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 read was clashing with the write.

Although in many ways this model of bits captures the behaviour that much formal analysis of wait-free protocols is built upon, there are at least three ways in which it is open to challenge. The first, and perhaps least significant, observation is that hardware bits can be engineered so that when they are overwritten with the same value that they already hold, the latched value does not flicker. The second (and probably the most significant) observation is that when the operating constraints that apply to most digital electronic circuits are taken into account, it is not feasible for multiple reads to clash with the changing bit value. This is because in edge-triggered circuits, the transition time for a latch to change (in response to the leading edge of the write signal) is independent of the duration of the write, and is short with respect to the set-up-and-hold and propagation times of the circuit. In other words, in such circuits at most one read can clash with the changing value. This behaviour is captured in later bit models that we present. The third observation is that BIT_{fm} fails to model the fact that shared bits can enter a metastable state when they are latched while the value being written into them is changing.

4.3 BIT_{sm}: Stable, Multiple-Clash Bit Model

This model modifies the previous by ensuring that the value of a bit is not disturbed when it is overwritten with the same value as it already holds. In this paper we call such a bit "stable". If care is taken it is possible to engineer digital circuits to behave in this way.

```
BITsm(val) = sw?x -> (if x == val then
                         BITsm_w_stable(val)
                      else
                         BITsm_w(val, x)) []
             sr -> BITsm_r(val)

BITsm_w(val, x) = ew -> BITsm(x) []
                  sr -> BITsm_wr(val, x)

BITsm_r(val) =
   sw?x -> (if x == val then
               BITsm_wr_stable(val)
            else
               BITsm_wr(val, x)) []
   er!val -> BITsm(val)

BITsm_wr(val, x) =
   ew -> BITsm_r_clashed(x) []
   (er!b0 -> BITsm_w(val, x) |~|
    er!b1 -> BITsm_w(val, x))

BITsm_r_clashed(val) =
   sw?x -> BITsm_wr(val, x) []
   (er!b0 -> BITsm(val) |~|
    er!b1 -> BITsm(val))

BITsm_w_stable(val) =
   ew -> BITsm(val) []
   sr -> BITsm_wr_stable(val)

BITsm_wr_stable(val) =
   ew -> BITsm_r(val) []
   er!val -> BITsm_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, BIT_{sm} is an L-safe register (i.e. it refines BIT_{fm}), but it does not itself exhibit all the L-safe behaviours. Lamport notes that stable bits are L-regular, [Lam86b].

4.4 BIT_{fs}: Flickering, Single-Clash Bit Model

This bit model is not stable, but it encodes the property that at most one read may clash with a write. This is a corollary of another practical engineering issue relating to shared bit variables that needs to be incorporated into our model. This issue challenges the erroneous assumption that for a variable to be an ACM, access at arbitrarily fast speeds must be possible. This is important because 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. Shared-variables that have minimum access times are still ACMs providing they do not force a synchronisation between the reader and writer accessing them. In particular, a maximum access frequency is compatible with relative asynchrony, and places no constraints on the ratio between the reader and writer frequencies.

It also needs to be appreciated that actual digital electronic circuit implementations of bits 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[12].

```
BITfs(val) = sw?x -> BITfs_w(val, x) []
             sr -> BITfs_r(val)

BITfs_w(val, x) = ew -> BITfs(x) []
                  sr -> BITfs_wr(val, x)

BITfs_r(val) = sw?x -> BITfs_wr(val, x) []
               er!val -> BITfs(val)

BITfs_wr(val, x) =
    ew -> BITfs_r_clashed(x) []
   (er!b0 -> BITfs_w_r_occured(val, x) |~|
    er!b1 -> BITfs_w_r_occured(val, x))

BITfs_r_clashed(val) = er!b0 -> BITfs(val) |~|
                       er!b1 -> BITfs(val)

BITfs_w_r_occured(val, x) = ew -> BITfs(x)
```

4.5 BIT_{ss}: Stable, Single-Clash Bit Model

This encodes both the stability and the single-clash constraints introduced in earlier models.

```
BITss(val) = sw?x -> (if x == val then
                         BITss_w_stable(val)
                      else
                         BITss_w(val, x))
```

[12] This assumes that switching and latching are triggered by the edge of the clock, and hence the switching and latching durations are independent of the writer's and reader's clock frequency. In other words, in this model the beginning and end of the switching duration are modelled by the *sw* and *ew* events, respectively; and the beginning and end of the latching duration are modelled by the *sr* and *er* events, respectively.

```
            [] sr -> BITss_r(val)

BITss_w(val, x) = ew -> BITss(x) []
                  sr -> BITss_wr(val, x)

BITss_r(val) = sw?x -> (if x == val then
                           BITss_wr_stable(val)
                        else
                           BITss_wr(val, x))
               [] er!val -> BITss(val)

BITss_wr(val, x) =
   ew -> BITss_r_clashed(x) []
   (er!b0 -> BITss_w_r_occured(val, x) |~|
    er!b1 -> BITss_w_r_occured(val, x))

BITss_r_clashed(val) = er!b0 -> BITss(val) |~|
                       er!b1 -> BITss(val)

BITss_w_stable(val) = ew -> BITss(val) []
                      sr -> BITss_wr_stable(val)

BITss_wr_stable(val) =
    ew -> BITss_r(val) []
    er!val -> BITss_w_stable(val)

BITss_w_r_occured(val, x) = ew -> BITss(x)
```

4.6 Modelling Metastability in Bit Models

None of the four models of shared bits given so far model metastability. However, metastability can be added to each of the above models, giving a total of eight different models of shared bits in CSP (plus the H-atomic model). Metastability is modelled by extending the alphabet of the channels associated with the *sw* and *er* events so that they include an extra dithering value, *d*. This value is returned should a read and write clash.

Rather than repeat each of the four models given above, we only illustrate the modelling of metastability by extending the BIT_{fm} model, resulting in the BIT_{fm_meta} model. The other models (BIT_{sm_meta}, BIT_{fs_meta}, and BIT_{ss_meta}) are modified similarly.

```
BITfm_meta(val) =
   sw?x -> BITfm_meta_w(val, x) []
   sr -> BITfm_meta_r(val)

BITfm_meta_w(val, x) =
   ew -> BITfm_meta(x) []
   sr -> BITfm_meta_wr(val, x)

BITfm_meta_r(val) =
   sw?x -> BITfm_meta_wr(val, x) []
   er!val -> BITfm_meta(val)

BITfm_meta_wr(val, x) =
   ew -> BITfm_meta_r_clashed(x) []
   (er!b0 -> BITfm_meta_w(val, x) |~|
    er!b1 -> BITfm_meta_w(val, x) |~|
    er!d -> BITfm_meta_w(val, x))

BITfm_meta_r_clashed(val) =
   sw?x -> BITfm_meta_wr(val, x) []
   (er!b0 -> BITfm_meta(val) |~|
```

```
    er!b1 -> BITfm_meta(val) |~|
    er!d -> BITfm_meta(val))
```

4.7 Modelling Metastability in the Algorithms

At this point it is necessary to consider how the values read from bit variables are handled in the reading process. For the four models that do not capture metastability, a simple process which reads a bit variable, B, and then uses (in some unspecified way) the value read would be:

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

However, when a shared bit such as B is modelled by a process which handles metastability, the value returned by a read of the bit may now be the metastable value, d. The variables (and the operations that are performed on them) in the reading algorithm have to be modified to handle this extra value. Furthermore, the algorithm needs to be able to model the fact that metastable values may decay into a standard binary value at some point in the future.

The way adopted here of systematically extending the local variables and enabling them to model metastability decay is to use a process in parallel with the reading process for each local bit variable in the process. Therefore, in the above example, the local bit variable "x" is modelled by a "local bit process". The value read into "x" from shared bit B is written into (and, subsequently, read out from) this process.

We present two different models of local bits, $LB1$ and $LB2$. LB1 can either autonomously resolve metastable values into binary ones (if it contains a metastable value), or it can be set or read.

```
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))
```

LB2 is similar, except that it models the case that metastable values resolve into binary ones before they are used by resolving any metastable value when it is written into it[13].

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

Resolving metastable values as they are set obviously ensures that all subsequent reads will get the same non-deterministically chosen value. 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, LB2 is a more appropriate model for a local bit variable than LB1. However, when such delays are not present, LB1 is more appropriate.

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

[13] In practice, of course, metastable circuits cannot be arbitrarily forced into a stable state, otherwise metastability would not be the fascinating phenomena and problem that it is. Nevertheless, this is an adequate CSP model of the behaviour of metastable values which resolve before being used.

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

Here the first two actions are reading the shared bit variable, B, as before; this value is then *set* in a local bit variable, B_LB. This is then read back into the reading process thus giving the opportunity for the potentially metastable value to resolve to a binary one, when the LB1 model is used. The value is then used as before, e.g. use1(x). 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 preceded with such a *get*, as this allows metastability resolution that occurs in the local variable process to influence the algorithm.

A further modelling issue arises for some of the ACMs considered here. It is necessary to model the "operation" of using a potentially metastable variable in a conditional statement which determines the flow of control of the algorithm. The following CSP process illustrates the kind construct found in some ACMs:

```
READER3 = B.sr -> B.er?x ->
          (if x == b1 then
                use1(x) -> READER3
           else use2(x) -> READER3)
```

It is not immediately obvious how such a situation should be modelled, when the shared bit can return the metastable value "d". One possibility is simply to treat "d" like a third value. (In other words, to leave the model as above.) In if-then-else statements, this has the effect of making the metastable value always favour the else branch. This seems implausible and means that possible behaviours will not be modelled. A better option is to change the model of the algorithm to explicitly check for "d" and when it is detected enter a new branch which involves a non-deterministic choice of the other branches. This is illustrated in READER4:

```
READER4 = B.sr -> B.er?x ->
          (if x == d
           then Reader4_Branch1 |~|
                Reader4_Branch2
           else
              (if x == b1
               then Reader4_Branch1
               else Reader4_Branch2))
```

It can be argued, however, that this is still too benign: it fails to model any indeterminacy about which value is loaded into the processor's next instruction register should the metastable value resolve during that operation. It seems conceivable that the two addresses might be merged to produce a (practically random) third address. Clearly, should this happen it is no longer possible to bound the potential behaviours. Such a catastrophic event needs to be considered, and is captured in our CSP model by making CHAOS a possible behaviour. This is illustrated in READER5:

```
READER5 = B.sr -> B.er?x ->
          (if x == d
           then Reader5_Branch1 |~|
                Reader5_Branch2 |~|
                CHAOS(Events)
           else
              (if x == b1
               then Reader5_Branch1
               else Reader5_Branch2))
```

Table 1: Descriptions of the Different Bit Models

Model	Model Description
BIT_{H_atomic}	H-atomic. Forces mutually exclusive access to each bit
BIT_{fm}	L-safe. Flickers on all over-writes. Multiple clashes allowed.
BIT_{sm}	Stable when overwritten with same value. Multiple clashes allowed.
BIT_{fs}	Flickers on all over-writes. Single clashes allowed.
BIT_{ss}	Stable when overwritten with same value. Single clashes allowed.
BIT_{fm_meta} LB1	As BIT_{fm} with re-readable metastability.
BIT_{fm_meta} LB2	As BIT_{fm} with read once metastability.
BIT_{sm_meta} LB1	As BIT_{sm} with re-readable metastability.
BIT_{sm_meta} LB2	As BIT_{sm} with read once metastability.
BIT_{fs_meta} LB1	As BIT_{fs} with re-readable metastability.
BIT_{fs_meta} LB2	As BIT_{fs} with read once metastability.
BIT_{ss_meta} LB1	As BIT_{ss} with re-readable metastability.
BIT_{ss_meta} LB2	As BIT_{ss} with read once metastability.

5 Specification Models

This section describes CSP models of data-coherence, sequencing and freshness. These are the formal models against which models of the ACMs will be checked for trace-refinement.

5.1 Data-Coherence

Data-coherence can be modelled abstractly by giving a process that is semi-regular – that is, a process in which non-clashing reads get the values stored in the variable, while clashing reads are constrained only to return one of the values that have been written into the ACM. This definition of data-coherence works for ACMs which are not conflict-free. Data-coherence for conflict-free ACMs can be specified by asserting that no buffer is ever read while it is being written or written while it is being read.

```
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))
```

The above specification works by the process keeping a set (vals) of the values written into it so far. A clashing read returns a non-deterministically chosen value from this set.

5.2 L-Regular

Fortunately, for ACMs which communicate finite values of types with finite domains, L-regularity is a finite state property. It is therefore possible to define a CSP process which models the

L-regular property, thus making (local) freshness something that can be determined by model-checking. See Section 3. 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, and it resets the set at appropriate points (such as when a write finishes when a read is not active, and when a read finishes when a write is not active).

```
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))
```

5.3 L-Atomicity

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.

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:

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

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

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

Core_H_Atomic_state =
    Core_H_Atomic_Var(var_name, 1)

LAtomicACM =
   (((Read ||| Write) [| {| var_name |} |]
               Core_H_Atomic_State) \ {| var_name |})
```

Table 2: Lamport's Regular ACM Analysis Results

Lamport's Regular ACM with all the control bits modelled as:	Data-Coherence (Semi-Regular)	L-Regular (Local Freshness)	L-Atomic (Global Freshness)
BIT_{H_atomic}	$\surd$	$\surd$	$\times$
BIT_{fm} (L-safe)	$\times$	$\times$	$\times$
BIT_{sm}	$\surd$	$\surd$	$\times$
BIT_{fs}	$\times$	$\times$	$\times$
BIT_{ss}	$\surd$	$\surd$	$\times$
BIT_{fm_meta} LB1	$\times$	$\times$	$\times$
BIT_{fm_meta} LB2	$\times$	$\times$	$\times$
BIT_{sm_meta} LB1	$\surd^*$	$\surd^*$	$\times$
BIT_{sm_meta} LB2	$\surd$	$\surd$	$\times$
BIT_{fs_meta} LB1	$\times$	$\times$	$\times$
BIT_{fs_meta} LB2	$\times$	$\times$	$\times$
BIT_{ss_meta} LB1	$\surd^*$	$\surd^*$	$\times$
BIT_{ss_meta} LB2	$\surd$	$\surd$	$\times$

5.4 Relationships Between the Specifications

We have achieved some validation of these specification models by showing that `SemiRegACM` is trace-refined by `RegACM`, and that `RegACM` is trace-refined by `LAtomicACM`, as one would expect. Furthermore, by composing LAtomicACM with (a) a writer which writes a monotonically increasing sequence of values, and (b) a reader which fails if it does not read a weakly monotonically increasing sequence, and (c) then checking that the reader does not fail, we have shown that LAtomicACM preserves the ordering of data passed through it. As mentioned above, this is another defining property of L-atomic ACMs.

6 Results of Model-Checking ACMs

This section reports on the model-checking a number of ACMs in the literature against each of the main specification properties (semi-regular, L-regular, L-atomic) for each of the nine shared bit models described in Section 4. Each protocol is briefly described according to the classification schemes of Section 3, and the results of the model-checking that ACM are tabulated.

Table 1 summarises the properties of each bit model.

6.1 Lamport's Regular ACM

In a seminal paper in 1986, [Lam86b], Lamport described various protocols, including a multivalued L-regular ACM. It is not a very practical ACM, For example, it requires n-bits to communicate values from a data type with cardinality n. However, the ACM by Kirousis et. al, [KKV87], is built on top of a multivalued L-regular ACM, and, in particular, they reference Lamport's implementation.

The results of analysing this multi-valued L-regular register are summarised in Table 2.

A number of things should be noted about these results. Firstly, the failures of the protocol when shared bits flicker are due to our adoption of a simplistic model of the algorithm given in [Lam86b]. The protocol would not fail in this way if, when shared bits already contain the value to be written, the write algorithm did not update them. Lamport himself advocates this scheme, [Lam86b].

Secondly, the protocol's correctness in the presence of metastability depends on how potentially metastable values in conditional statements are modelled, and is indicated by the $\surd^*$ symbols in Table 2. This is because Lamport's reader algorithm uses the value of a (potentially

Table 3: Simpson's 4-Slot Analysis Results

1987 4-Slot with all the control bits modelled as:	Data-Coherence (Semi-Regular)	L-Regular (Local Freshness)	L-Atomic (Global Freshness)
BIT_{H_atomic}	√	√	√
BIT_{fm} (L-safe)	√	×	×
BIT_{sm}	√	√	×
BIT_{fs}	√	×	×
BIT_{ss}	√	√	√
BIT_{fm_meta} LB1	×	×	×
BIT_{fm_meta} LB2	√	×	×
BIT_{sm_meta} LB1	×	×	×
BIT_{sm_meta} LB2	√	√	×
BIT_{fs_meta} LB1	×	×	×
BIT_{fs_meta} LB2	√	×	×
BIT_{ss_meta} LB1	×	×	×
BIT_{ss_meta} LB2	√	√	√

metastable) shared bit as the test for loop termination. When this is understood as being a non-deterministic choice between the two options of terminating the loop or continuing to execute it (as in READER4 above), the protocol remains L-regular. However, should this be modelled as allowing chaos (as in READER5 above), the protocol fails to be data-coherent.

6.2 Simpson's 4-Slot ACM

In 1987 Simpson described a "4-Slot" ACM, [Sim87] and [Sim90]. It is a buffer-based, conflict-free, 1-Writer 1-Reader write-once, read-once multivalued ACM. It is buffer-optimal because it only uses four buffers (slots). It is particularly simple: the write operation consists of only five actions, and read operation only four, and neither algorithm contains any branching. It has been the subject of various academic analyses, based on different assumptions and using various technologies and techniques: [Sim92, Sim97, BJA96, Bro99, XC00, Cla00, Xia00, Rus02, HP02] and [Hen03].

It should be noted that Haldar and Vidyshankar's impossibility result (mentioned above in Section 3) applies to the 4-Slot ACM. In other words, Simpson's 4-slot cannot be L-atomic if its shared bits are L-safe registers. Rushby's work, [Rus02], establishes that the 4-Slot does indeed fail to be L-atomic when its shared bits are L-safe. However, with models of bits which are arguably more realistic than the L-safe model (as they assume well-handled metastability and typical timing constraints), the 4-Slot ACM *is* L-atomic. The full results of analysing Simpson's 1987 4-Slot mechanism are summarised in Table 3. Note, row 2 of this table confirms Rushby's results, [Rus02].

Simpson's original 1987 4-Slot has therefore been shown to be L-atomic (and hence data-coherent and globally fresh) when modelled with the BIT_{ss} and BIT_{ss_meta} LB2 shared bit models. In other words, it is an L-atomic variable providing that its bits are engineered to be flicker-free variables, and the bits are implemented in hardware in such a way that at most one read of the bit can clash with any write of that bit. As one would expect, this result is not changed by metastability which decays into a binary value before being used (the LB2 rows).

An important ramification follows from both: (a) the 4-Slot being L-atomic with BIT_{ss} bits; and (b) it not even being L-regular for BIT_{fm} (L-safe) bits. Since BIT_{ss} is a perfectly reasonable requirement to place on hardware implementations of a bit (as discussed in Section 4) the analysis of protocols based on L-safe models of shared bits may be pessimistic. This raises questions about the practical relevance of L-atomic impossibility and optimality results that are based on the assumption that bits are L-safe variables.

Table 4: KKV's ACM Analysis Results

The KKV ACM with all the control bits modelled as:	Data-Coherence (Semi-Regular)	L-Regular (Local Freshness)	L-Atomic (Global Freshness)
BIT_{H_atomic}	?	×	×
BIT_{fm} (L-safe)	?	×	×
BIT_{sm}	?	×	×
BIT_{fs}	?	×	×
BIT_{ss}	?	×	×

Perhaps an even more important ramification arises from the fact that although the 4-Slot maintains data-coherence with L-safe bits, it fails to do so for certain plausible models of metastability (the LB1 rows). This means that analysis of protocols based on L-safe models of shared bits may be unduly optimistic. This therefore calls into question all analysis based on the L-safe bit register assumption, which has been a common assumption made in the literature.

It is interesting to note that, alone of the protocols considered here, Simpson's 4-Slot ACM does not require any use of potentially metastable values to determine the control-flow of either the reader or writer algorithms. However, using potentially metastable values to determine the value to be read may result in random memory locations be accessed with unknown consequences.

6.3 Kirousis, Kranakis and Vitányi's ACM

Kirousis, Kranakis and Vitányi's introduced a protocol which they claim is L-atomic in [KKV87]. It is a multivalued, buffer-based, conflict-free, write-once and read-once ACM. It has four buffers, so it is buffer-optimal. However, it is not immediately obvious how many shared bits that it uses, as it exploits three shared L-regular registers which can hold thirteen different values. Kirousis et. al. suggest that these be implemented using Lamport's multi-valued regular registers proposed in [Lam86b]. The results in Table 4 have been obtained by building the KKV protocol out of Lamport L-regular registers.

Assuming we have understood and modelled this protocol properly, we have shown that this relatively complex ACM is flawed. It fails to be conflict-free for the initialisations given in the paper, [KKV87], for any of the bit models. However, when the control variables are made L-atomic, and an initialisation is adopted which is compatible with ongoing behaviour, we have shown that the protocol is L-atomic. This, however, is a very weak result, as it only shows that this protocol implements a multi-valued L-atomic ACM when it is built out of multiple multi-valued L-atomic ACMs. Unfortunately, the table is incomplete because we have not been able to determine whether the protocol preserves data-coherence for any of the bit models. Twin Pentium IV processors running at 3GHz for 3 weeks with over 35Gb of disk space failed to produce a result.

Our results emphasise the importance of developing machine-checked proofs of correctness of ACMs, rather than relying on proof sketches. An informal 'proof sketch' or 'rigorous argument' carries little conviction when performed for such complex protocols.

6.4 Tromp's Atomic Bit ACM

In [Tro89], Tromp gives a protocol for an "atomic bit". It is a buffer-based protocol which consists of three shared bits, one of the shared bits, "V", being the buffer. Tromp provides a rigorous proof that it is indeed L-atomic, [Tro89], however, this proof clearly fails to take metastability into account. The results of model checking Tromp's Atomic Bit using the same bit models introduced above are shown in Table 5.

Table 5: Tromp's Atomic-Bit Analysis Results

Tromp's Atomic Bit ACM with all the bits modelled as:	Data-Coherence (Semi-Regular)	L-Regular (Local Freshness)	L-Atomic (Global Freshness)
BIT_{H_atomic}	$\surd$	$\surd$	$\surd$
BIT_{fm} (L-safe)	$\surd$	$\surd$	$\surd$
BIT_{sm}	$\surd$	$\surd$	$\surd$
BIT_{fs}	$\surd$	$\surd$	$\surd$
BIT_{ss}	$\surd$	$\surd$	$\surd$
BIT_{fm_meta} LB1	$\surd$*	$\surd$*	$\times$
BIT_{fm_meta} LB2	$\surd$	$\surd$	$\surd$
BIT_{sm_meta} LB1	$\surd$*	$\surd$*	$\times$
BIT_{sm_meta} LB2	$\surd$	$\surd$	$\surd$
BIT_{fs_meta} LB1	$\surd$*	$\surd$*	$\times$
BIT_{fs_meta} LB2	$\surd$	$\surd$	$\surd$
BIT_{ss_meta} LB1	$\surd$*	$\surd$*	$\times$
BIT_{ss_meta} LB2	$\surd$	$\surd$	$\surd$

It should be noted that the second row confirms Tromp's analysis that his protocol is L-atomic when bits are assumed to be L-safe, i.e. when there is no metastability. However, it is possible for this algorithm to return a metastable value when unconstrained metastability may occur. If one accepts that such values are (ultimately) binary ones and hence should be modelled as a non-deterministically chosen binary value, it can be concluded that Tromp's bit algorithm is L-regular even in the presence of re-readable metastability (the LB1 rows). However, if such values are considered to be a third (non-binary) one, the protocol does not even preserve data-coherence (hence the $\surd$* in the table).

It should be noted, however, that when re-readable metastability is present, this protocol fails to preserve sequencing, and hence fails to be L-atomic. This situation corresponds to the read algorithm being executed so fast that metastable values may be re-read inducing further metastability throughout the algorithm. This confirms the danger of omitting metastability from protocol analysis.

6.5 Tromp's First 4-Track ACM

In [Tro89], Tromp gives a 4-Track ACM which he builds on top of his atomic bit ACM. It is a buffer-optimal buffer-based, conflict-free, write-once, read-once multivalued ACM built from four L-atomic bits (or, expanding his atomic bit ACMs, twelve shared bits). He provides a rigorous argument that it is L-atomic.

The results of model checking Tromp's First 4-Track ACM using the same bit models are shown in Table 6. The model was constructed using a model of Tromp's atomic bit protocol analysed above.

This shows that this protocol nominally works in the presence of re-readable metastability (the LB1 rows). This is surprising, given that Tromp's paper gives no hint that metastability was considered when the protocol was developed. It is also surprising, because it builds upon Tromp's atomic bit protocol, which itself fails to be L-atomic in the presence of re-readable metastability.

It will be noted, however, that the results for all the LB1 rows are again marked with a star. This is because Tromp's algorithm makes use of a variable which may be metastable when it is used in a conditional check. When this is modelled as a non-deterministic choice between the stated branches, the protocol works; but when an arbitrary alternative behaviour might result (modelled by the CHAOS process), not surprisingly, the protocol fails. (See the discussion in Section 4.7 above.)

Table 6: Tromp's First 4-Track ACM Analysis Results

Tromp's Basic 4-Track ACM with all the bits modelled as:	Data-Coherence (Semi-Regular)	L-Regular (Local Freshness)	L-Atomic (Global Freshness)
BIT_{H_atomic}	√	√	√
BIT_{fm} (L-safe)	√	√	√
BIT_{sm}	√	√	√
BIT_{fs}	√	√	√
BIT_{ss}	√	√	√
BIT_{fm_meta} LB1	√*	√*	√*
BIT_{fm_meta} LB2	√	√	√
BIT_{sm_meta} LB1	√*	√*	√*
BIT_{sm_meta} LB2	√	√	√
BIT_{fs_meta} LB1	√*	√*	√*
BIT_{fs_meta} LB2	√	√	√
BIT_{ss_meta} LB1	√*	√*	√*
BIT_{ss_meta} LB2	√	√	√

6.6 Tromp's Efficient 4-Track ACM

In [Tro89], Tromp also presents a variant of his 4-Track ACM which uses fewer shared bits – only eight compared with twelve. It is not built from Tromp's Atomic Bit protocol, but exploits the fact that the reader and writer cannot clash on every shared bit. The results of model-checking Tromp's Efficient 4-Track ACM using the same bit models are shown in Table 7.

Table 7: Tromp's Efficient 4-Track ACM Analysis Results

Tromp's Revised 4-Track ACM with all the bits modelled as:	Data-Coherence (Semi-Regular)	L-Regular (Local Freshness)	L-Atomic (Global Freshness)
BIT_{H_atomic}	√	√	√
BIT_{fm} (L-safe)	√	√	√
BIT_{sm}	√	√	√
BIT_{fs}	√	√	√
BIT_{ss}	√	√	√
BIT_{fm_meta} LB1	√*	√*	√*
BIT_{fm_meta} LB2	√	√	√
BIT_{sm_meta} LB1	√*	√*	√*
BIT_{sm_meta} LB2	√	√	√
BIT_{fs_meta} LB1	√*	√*	√*
BIT_{fs_meta} LB2	√	√	√
BIT_{ss_meta} LB1	√*	√*	√*
BIT_{ss_meta} LB2	√	√	√

These results show that Tromp's Efficient 4-Track ACM has similar behaviour to his original 4-Track ACM; it is equally as impressive in handling all kinds of metastability, but is open to the same possibility of failing with arbitrary behaviour in the presence of re-readable metastability.

6.7 Haldar and Subramanian's ACM

Haldar and Subramanian introduced a multivalued, buffer-based, ACM in [HS94]. It is buffer-optimal, and only used four shared bits. However it is not write-once. It is therefore not a counter-example to Haldar and Vidyasankar's impossibility result mentioned above, [HV01].

The results of model-checking Haldar and Subramanian's ACM using the same bit models are shown in Table 8.

Table 8: Haldar and Subramanian's ACM Analysis Results

Haldar & Subramanian's ACM with all the bits modelled as:	Data-Coherence (Semi-Regular)	L-Regular (Local Freshness)	L-Atomic (Global Freshness)
BIT_{H_atomic}	√	√	√
BIT_{fm} (L-safe)	√	√	√
BIT_{sm}	√	√	√
BIT_{fs}	√	√	√
BIT_{ss}	√	√	√
BIT_{fm_meta} LB1	×	×	×
BIT_{fm_meta} LB2	√	√	√
BIT_{sm_meta} LB1	×	×	×
BIT_{sm_meta} LB2	√	√	√
BIT_{fs_meta} LB1	×	×	×
BIT_{fs_meta} LB2	√	√	√
BIT_{ss_meta} LB1	×	×	×
BIT_{ss_meta} LB2	√	√	√

The protocol works for all our bit models, except those that support re-readable metastability (the LB1 rows). This again underlines the importance of ensuring that metastability is well handled in a given implementation of an ACM.

7 Conclusions

This paper has shown that metastability and other hardware timing constraints and behaviour can be incorporated into models used to analyse asynchronous wait-free protocols. Furthermore, it has shown that there are strong reasons for not continuing to use L-safe registers as a model of shared bits in such algorithms. In summary, these include the facts that:

1. the analysis may be pessimistic (the 4-Slot ACM was shown not to be L-regular, although it is even L-atomic under stronger but realistic assumptions);

2. the analysis may be optimistic (all the ACMs were shown to have failure modes when metastability is not contained); and

3. overly complex and inefficient ACMs may be adopted, when smaller and more efficient ACMs could be used. The 4-Slot ACM uses fewer shared bits than it is possible to build an ACM with that is conflict-free, and write and read-once, assuming Haldar and Vidyasankar's unpublished impossibility result is true, [HV01].

It is not possible to conclude that any one ACM is better than all its rivals in all circumstances from these results. Haldar and Subramanian's ACM and Simpson's 4-Slot use fewer control bits than Tromp's Efficient ACM – four rather than eight. The cost for Haldar and Subramanian's ACM is that it is not write-once; this might have unacceptable performance implications when large data-structures are being communicated. The cost for Simpson's 4-Slot is that it only works when bits are implemented so that they are stable and at most one clash may occur while a bit is being set. In the large class of implementations where this is true, Simpson's 4-Slot will be correct and more efficient than Tromp's. All the ACMs have difficulties when they are executed so fast that metastable values may be re-read before they resolve into binary values. However, Simpson's 4-Slot ACM does not exhibit the failure-mode by which a metastable value may directly upset control flow. In different cases, where BIT_{ss} assumptions do not hold, Tromp's Efficient 4-Track algorithm is the most comprehensive and efficient alternative considered here.

This work has underlined that it would be desirable to develop a collection of impossibility results for wait-free protocols based on stronger models of bits such as the BIT_{ss} model.

Acknowledgements

MBDA UK Ltd. and the BAE SYSTEMS DCSC funded this research.

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.

[BP89] James E. Burns and Gary L. Peterson. The Ambiguity of Choosing. In *Proceedings of 8^{th} Annual Symposium on Principles of Distributed Computing (PODC'89)*, pages 145–157. ACM Press, 1989.

[Bro99] P.J. Brooke. *A Timed Semantics for a Hierarchical Design Notation.* PhD thesis, Department of Computer Science, University of York, April 1999.

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

[Cla00] I.G. Clark. *A Unified Approach to the Study of Asynchronous Communication Mechansims in Real-Time Systems.* PhD thesis, London University, King's College, May 2000.

[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.

[Gin03] Ran Ginosar. Fourteen Ways to Fool Your Synchronizer. In *Proceedings of ASYNC'03*, pages 89–95, 2003.

[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.

[Hen03] N. Henderson. Proving the Correctness of Simpson's 4-Slot ACM Using An Assertional Rely-Guarantee Proof Method. In *FM 2003: The 12^{th} International FME Symposium*, Lecture Notes in Computer Science. Springer, 2003.

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

[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.

[HS94] S. Haldar and P.S. Subramanian. Space-Optimum Conflict-Free Construction of Buffer-Optimal 1-Writer 1-Reader Multivalued Atomic Variable. In *Proceedings of 8th International Workshop on Distributed Algorithms (WDAG'94)*, number 857 in Lecture Notes in Computing, pages 116–129. Springer, 1994.

[HV95] S. Haldar and K. Vidyasankar. Buffer-Optimal Constructions of 1-Writer Multireader Multivalued Atomic Shared Variables. *Journal of Parallel and Distributed Computing*, 31:174–180, 1995.

[HV01] S. Haldar and K. Vidyasankar. Space-Optimal Buffer-Based Conflict-Free Construction of 1-Writer 1-Reader Multivalued Atomic Variables from Safe Bits. Unpublished Paper, 2001.

[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.

[KBY02] David J. Kinniment, Alexandre Bystrov, and Alex V. Yakovlev. Synchronization Circuit Performance. *IEEE Journal of Solid-State Circuits*, 37(2):202–209, 2002.

[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.

[KKV87] L.M. Kirousis, E. Kranakis, and P.M.B. Vitányi. Atomic Multireader Register. In *Proceedings of the Workshop on Distributed Algorithms*, number 312 in Lecture Notes on Computer Science, pages 278–296. Springer, 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.

[Pet83] Gary L. Peterson. Concurrent Reading While Writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, January 1983.

[PHA04] S.E. Paynter, N. Henderson, and J.M. Armstrong. Ramifications of Metastability in Bit Variables Explored via Simpson's 4-Slot Mechanism. *To Appear in Formal Aspects of Computing*, ?(?):?–?, 2004.

[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.

[Sim97] H.R. Simpson. Role Model Analysis of an Asynchronous Communication Mechanism. *IEE Proceedings of Computers and Digital Techniques*, 144(4):232–240, July 1997.

[Tro89] John Tromp. How to Construct an Atomic Variable (Extended Abstract). In J. Bermond and M. Raynal, editors, *Proceedings of the 3^{rd} International Workshop on Distributed Algorithms*, number 392 in Lecture Notes in Computer Science, pages 292–302. Springer, 1989.

[XC00] 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. In *Hardware Design and Petri-Nets*, pages 33–50. Kluwer Academic Publishers, 2000.

[Xia00] Fei Xia. *Supporting the MASCOT Method with Petri Net Techniques for Real-Time Systems Development.* PhD thesis, London University, King's College, January 2000.