Quantitative modelling of asynchronous variables

Fei Xia and Ian Clark

School of EEE, Newcastle University, NE1 7RU, UK
fei.xia@newcastle.ac.uk, ian.clark@newcastle.ac.uk

Abstract. Variables being passed between processes not synchronized for the communication may be affected by the lack of synchrony between the processes and such passing of variables may also affect the nominal asynchrony between the communicating processes. There exists a large body of research on the data communication between asynchronous processes exemplified by Lamport's atomic registers and Simpson's multi-slot asynchronous communication mechanisms (ACMs). Many of the existing solutions try to reduce the effects of the fundamental problem by reducing the timing independence to variables of very small size. For instance, Boolean and ternary control variables have been used to protect the usually larger data structures being passed. However, ultimately, the control variables must deal with the asynchrony between the communicating processes in some way. A Boolean variable (single bit) between an asynchronous reader-writer pair cannot avoid metastability or mutual exclusion protection, for instance. Existing models using formalisms such as Petri nets and process algebra and solutions based on state-space analysis provide a very good understanding in the qualitative behavior of such variables. In this paper we aim to expand this understanding to the quantitative by developing models in stochastic activity networks (SANs) with which quantitative investigations may be made with regard to such variables.

Keywords: asynchronous data communication, metastability, stochastic activity networks.

1 Introduction

In digital systems, before the entire world's systems can be synchronized on the same truly global clock, inevitably it would be necessary to communicate outside a particular clock domain. With continued increase of VLSI integration, the physical size of clock domains have become smaller, not larger, and the overall number of clock domains has also increased. For instance, whereas it used to be that everything on the motherboard of a desktop computer ran off a single global clock, now within a single chip there tends to be multiple clock domains as a single chip packs more computation power than multiple classical computers.

Crossing clock domain boundaries with data can be implemented in many ways. It can be done fully synchronously, through the temporary synchronization between two clock domains for the duration of data transfer. This can be found in many globally asynchronous locally synchronous (GALS) solutions where stretchable and/or pausible clocks are used [1]. It can also be done with a certain degree of asynchrony between the communicating processes, for instance through the use of data buffers. This can be found in the majority of network communications including networks on chip (NoC) solutions [2].

One of the ultimate examples of asynchronous data communication is the so-called fully asynchronous communication where there is no synchronization either actively administered (e.g. the use of GALS-style synchronization) or implied (e.g. through the buffer full or empty states, or the mutual exclusion/critical section protection of data), which ideally allows the processes to possess full temporal independence not affected by the act of communication. Lamport's atomic register followed by Simpson's multi-slot ACMs attempt to solve this problem [3, 4].

From this wide spectrum of problem statements and solutions, it is clear that there exist two fundamental desirable properties. These are:

- Asynchrony: Minimal obliged waiting for either the reader or the writer processes. Fully asynchronous communication aims for zero waiting on either side.
- Data transfer: Maximal quality for the data eventually read. This is usually described by a number of metric parameters, such as data coherence, data freshness, data sequencing, etc. and is different for different application scenarios. An intuitive understanding of data coherence, for instance, is that the writer, or anything else, should not be allowed to corrupt half-read data.

And the large number of existing solutions arrive at various trade-off points between these two qualities [8].

A substantial amount of research exists in this field, with a large number of attempts at provide qualitative modelling so that a solution may be tested for whether it violates data coherence, process asynchrony or any other metric and if so under what circumstances [8].

However there has been a total absence of any quantitative modelling method with which different solutions may be more precisely placed relative to each other in a quantitative map of trade-off.

1.1 Contributions and organization

This work is the first attempt at achieving quantitative models of asynchronous data communication. The language chosen is SANs, which provides opportunities of properly representing such phenomena as metastability quantitatively according to well accepted models [9]. Given that many solutions remove the problems caused by inter-process asynchrony away from potentially large data to usually small control variables, this work concentrate on modelling Boolean and ternary variables and their usual implementation using hardware latches.

The rest of the paper is organized as follows: Section 2 introduces the concept of the asynchronous variable, and describes quantitative models for the two essential properties for Boolean asynchronous variables. Section 3 describes more complex asynchronous variables, their implementation and modelling. Section 4 describes case studies where the models of asynchronous control variables are used to derive behaviors of larger systems in which they are used. Section 5 concludes the paper.

2 The Boolean asynchronous variable

Fig. 1 shows the basic concept of two asynchronous processes intercommunicating with one (writer) providing the data and the other (reader) making use of it. This is both a general description of all such data communications and a specific description of the passing of control variables. The difference is in what the data is and how it is meant to be used.

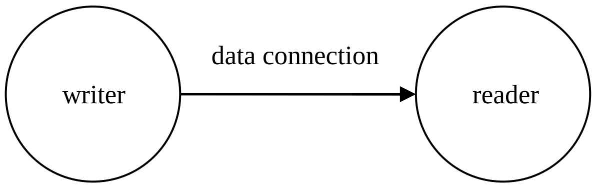

Fig. 1 Unidirectional inter-process data communication.

In a control variable situation, it may happen that the writer of the overall data communication may be the reader of a specific control variable, which is written by the reader process in the overall communication. In other words, although a specific instance of asynchronous data communication is usually defined as unidirectional, i.e. data passing from the writer to the reader, to support this communication some of the control variables may go the other way, to allow the reader to inform the writer of its current state, for instance. In the rest of the paper, unless otherwise stated, 'reader' and 'writer' pertain to the variable being discussed, which in most cases are binary and ternary control variables and not the main data.

It is clear from Fig. 1 that we assume each individual control variable to be unidirectional and cannot be written to by both sides. This simplifies the problem without limiting the solution space, as demonstrated by numerous existing work.

The smallest control variable is the smallest digital variable, i.e. a single binary bit. This is the subject of this section.

Such a variable can be transmitted from the writer to the reader in a number of ways:

(a) Fully synchronously: The reader and the writer need to be synchronized for a single clock period during which the reader directly reads a copy of this Boolean value from the writer's bus or output port. There needs to be no shared memory, just shared wires [5].

(b) With a single shared memory location to provide some degree of asynchrony: The single space FIFO buffer may be guarded with a MUTEX making it accessible by one process at any time [5].

(c) With an unguarded single space FIFO buffer: A fully asynchronous solution allows buffer access by both sides at the same time [4, 8].

All three methods face the following two issues:

- Metastability: The phenomenon of metastability, where a nominally Boolean signal takes a value that is neither 0 nor 1 which nevertheless may persevere for non-trivial amounts of time, is inevitable when you have two independently timed processes accessing the same memory element at the same time in certain conflicting ways, such as reading and writing at the same time or making requests to a MUTEX element at the same time [6]. The simplest 1-bit memory is a latch and synchronizers, MUTEXes and Boolean variables are implemented using circuits which could be classified as some type of latch. The first method therefore cannot avoid metastability at the synchronizer, the second method must face it at the MUTEX, and the third directly on the data bit.
- The relative timing of access from both sides is not specified. Reading may take the same time as, or a radically different time from, writing. And this potentially has an impact on the behaviors of all three methods.

The effects of these challenges on the different methods may be reasoned about qualitatively using existing research results and techniques [7]. For example, even though the fully asynchronous solution may sound unsafe because the metastability is on the data and not controlled by a MUTEX, or mitigated by multi-flop synchronizers, for a lot of control variables used in asynchronous data communications this causes, in practice, some non-deterministic delay [8]. Since the variable being communicated is a binary bit, the worst case scenario, i.e. the reader and the writer both accessing it at the same time, is that metastability may happen. Pragmatically, it is sensible to assume that once metastability happens, the variable eventually settles non-deterministically to one of the digital values: either 0 or 1. Metastability can only happen if the writer is in the process of changing the value of the bit when the reader attempts to access it. Hence either one of the settled values should be valid, for any sensible communication algorithm and implementation. The only thing the designer need to do is to make sure that the control variable is used after some time of its reading to provide it with enough probability to settle before use, as determined by the mean time between failure (MTBF) requirement of the design.

However, when a designer is making a decision on choosing one method over another, a quantitative exploration may be desirable in addition to qualitative considerations.

In this section, we develop quantitative models for both challenges, metastability and independent timing of reader and writer processes.

2.1 Quantitative modelling of metastability in a Boolean variable

To study the metastability behavior of a binary bit being passed from one asynchronous process to another, we assume it is implemented in the way described by Fig. 2. This is the passing of a binary variable from the writer to the reader, such that the reader's input variable *y* takes on the value of the writer's output variable *x* when the clock/control signal *cl* is set. In other words, *cl*: *y*=*x*.

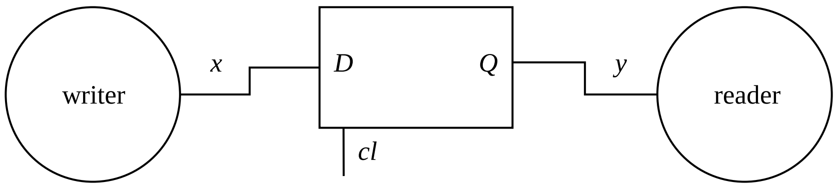

Fig. 2 A binary asynchronous variable.

In order to have metastability in a latch of this type, the clock or control signal *cl* is usually activated by some entity not temporally related to the writer, in other words, the signals *x* and *cl* may change very close in time causing metastability to happen at signal *y*. This directly corresponds with method (c) described above as a latch like this forms the unguarded FIFO buffer used in that method. On the other hand, since synchronizers are constructed out of essentially the same kind of circuit with the same metastability behavior, we can describe the metastability encountered by method (a) using the same technique.

A binary variable that may become metastable, and settling out of metastability

Representing the metastable value of a nominally Boolean variable as a distinct marking allows the convenient tracking of metastability and its effects. The SAN model of a Boolean variable that may become metastable is shown in Fig. 3.

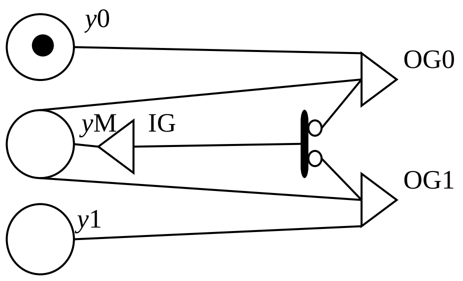

Fig. 3 SAN model of a Boolean variable *y* that may become metastable, including the process of metastability settling.

The process of the natural settling of metastability is usually regarded as stochastic with an average speed entirely dependent on the hardware implementation of the variable. This can be represented by a timed transition whose firing takes the value of the variable to 0 or 1 based on pre-determined probabilities. Established theory on metastability describes the settling as following an exponential process [5]. This can be represented by the timed transition having an exponential timing distribution whose mean rate λ can be found through hardware experimentation [5]. It is usual practice to assume that metastability settles to 0 or 1 with the same probability, i.e. 50%. However, the model in Fig. 3 allows arbitrary pairs of probabilities p_0=1-p_1 to be chosen, if experiments on hardware show a bias in one way or the other.

Coincidentally, for the purpose of using this model for analysis, exponential timing in the timed transition means that this part of the model does not introduce anything non-Markovian. An entirely Markovian system model usually allows not only simulations but also analytical reasoning [9].

The logic of the input and output gates in the model is defined as follows:

Gate	**Predicate**	**Function**
IG	`Mark(yM)==1`	

OG0		`Mark(yM)=0;`
		`if (Mark(y0)==0 and Mark(y1)==0)`
		`then Mark(y0)=1;`

OG1		`Mark(yM)=0;`
		`if (Mark(y0)==0 and Mark(y1)==0)`
		`then Mark(y0)=1;`

The settling countdown starts immediately when the variable enters metastability. At the end of the model-determined settling time, the marking in place *y*M is set to zero. However, when updating the variable to a digital value, the model needs to determine whether the settling process is at this moment still in charge of the value of the variable – it is entirely possible that during the expected duration of metastability settling time, when the settling transition is in the process of firing, the variable has otherwise been set to a secure digital value through other means such as having been successfully assigned a value by another operation. The functions of the output gates ensure that only when no such thing has happened (i.e. both digital places still have the marking of 0) the completion of the settling transition would set the expected digital value. Otherwise nothing is done as at the end of the expected settling time, the variable has already otherwise achieved a secure digital value.

Actively changing the variable value

In addition to the settling of metastability, which is a passive process, the value of a nominally Boolean variable may also be changed actively. An example of this is the setting of variable *y* to the value of variable *x* in Fig. 2.

To model metastability and its effect fully, we need to consider the following situations:

- When *cl* comes, *x* is stable at either 0 or 1 → *y* takes the value of *x*
- When *cl* comes, *x* is itself metastable, i.e. having the value of M → *y* has a probability of becoming M;
- When *cl* comes, *x* is being changed → *y* has a probability of becoming M;

This means that we need a place or places whose marking(s) indicate that *x* is being changed between the two digital values 0 and 1. In addition, *cl* itself is a signal whose change may take some time. This is best represented by having any *cl* change indicated by a marking in a place.

The SAN formalism facilitates the compact representation of such conditional relationships, once such states as ‘*x* is being changed’ and ‘*cl* is coming’ are represented by markings. Similar to the metastability settling speed, the probabilities of *y* getting a value of 0, M or 1 in any particular situation may be obtained through hardware experiments. Hardware characterization is the best method for generating the quantitative parameters for these models.

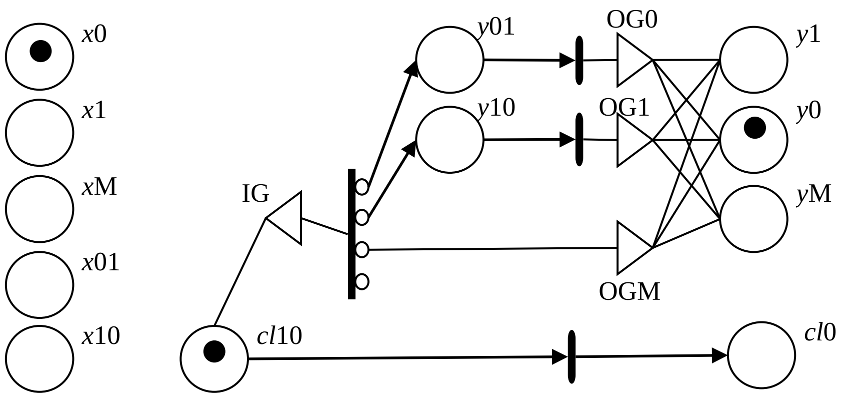

Fig. 4 Changing the value of an asynchronous Boolean variable.

Fig. 4 shows the structure of the SAN model of changing the value of a Boolean variable implemented using a circuit of the type shown in Fig. 2. Places named *x*0, *y*M, *cl*0, etc. denote that the signal takes a particular value. Places named with the signal name followed by two values denote that the signal is transitioning from the first value to the second, i.e. *x*01 denotes *x*: 0→1.

The logic of the input and output gates is as follows:

Gate	**Predicate**	**Function**
IG	`Mark(cl10)==1`	
OG0		`Mark(y=M)=0;` `Mark(y=0)=0;` `Mark(y=1)=1;`
OG1		`Mark(y=M)=0;` `Mark(y=1)=0;` `Mark(y=0)=1;`
OGM		`Mark(y=0)=0;` `Mark(y=1)=0;` `Mark(y=M)=1;`

The process of changing the value of *y* starts when *cl* is being reset (falling edge trigger on the clock, place *cl*10 marked). At the end of the process, the value of *y* is set according to what branch of the SAN the model has been progressing. The first transition on the left hand side is where the logic is that determines how the value of *y* will be set, and as such must correctly specify the probabilities of each of its branches, changing from 0 to 1, changing from 1 to 0, setting to M, and do nothing (keeping the old value of *y*).

This depends on the markings of the places listed on the left hand side of Fig. 4.

For instance, case 4 of the activity, do nothing, has a probability of 1 when *x*=0 and *y*=0, and a probability of 0 when *x*=0 and *y*=1. The probability of case 3 is non-zero if one of the places *x*M, *x*01 and *x*10 are marked. The probability of case 3 under different conditions when it is not zero can be determined through hardware characterization experiments [5].

This model has a relatively low precision as it assumes a constant probability of *y* entering metastability if, when *cl* changes, *x* is in the process of change. However, accepted theory of metastability indicates that the probability of *y* entering metastability is related to how close the *cl* and *x* changes are in time [5]. Assuming the same probability for all cases of overlapping access may not be precise enough for certainly analysis.

Timing issues

The precision of the above model can be improved by representing one of the changing processes, either *x* changing or *cl* falling, as constituting multiple steps. This extended representation is shown in Fig. 5.

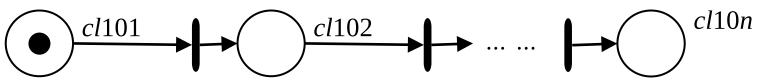

Fig. 5 Dividing a value change into multiple steps.

During a change of the signal *cl*, how much this change has progressed is indicated by where the token is in the chain. Usually variable value changes take deterministic time related to the speed of the hardware in which the process that makes this change is mapped. Such a deterministic delay can easily be divided into a sequence of deterministic delays as represented by the timed transitions in Fig. 5. For stochastic delays, this method is less effective and accurate unless the distributions of the sequence of smaller delays can be derived from the distribution of the overall delay, which is not always possible. In other words, if metastability is involved in any signal value change, this method of dividing into multiple smaller changes may be less effective and using a single probability as in the previous section may be unavoidable.

Knowing where a signal is in its changing process when the other signal is also changing allows the distance in time between these two changes to be represented, up to the precision of the stages in the model shown in Fig. 5. The model can therefore be made as precise as the time resolution of the hardware characterization data.

The other timing issue, that of the relative durations of time each of the reader and writer processes makes accesses to a shared resource, is automatically represented in models developed according to the methods given in this section. Together, the metastability settling time, rising time, and falling time of each signal fully describe overall process lengths such as how long a shared variable is accessed by the writer or the reader.

2.2 MUTEX and arbitration

For method (b), which protects the shared memory location with a mutual exclusion arrangement avoiding simultaneous accesses by both the reader and writer. The metastability and timing modelling can be derived based on the models presented in Section 2.1. This is because MUTEXes are usually constructed out of similar circuits, i.e. a single bit memory. However, since the input signals to MUTEXes are not the somewhat asymmetric data and clock with different functions, but fully symmetric requests, the models need to be modified to reflect this.

A typical MUTEX consists of an SR-latch followed by a metastability resolver. It functions as follows:

- Be ready to receive requests from two different processes when no grant is outstanding;
- Issue a grand to the process which has just produced a request;
- Withdraw a grant after receiving a reset of a request signal – a requesting process is assumed to reset its request once the granted resource has been made use of;
- When requests from both processes arrive close together, the latch may go into metastability, but the metastability resolver makes sure that no grants will be issued until the metastability has been resolved;
- A requesting process is assumed to hold up its request until a grant is issued – there is no withdrawal of requests without grants.

The model for the metastable state and its settlement is similar to Fig. 3, if a Boolean variable is used to issue grants, for instance y=0 grants process 0 and y=1 grants process 1, as is normal for MUTEX arbitration.

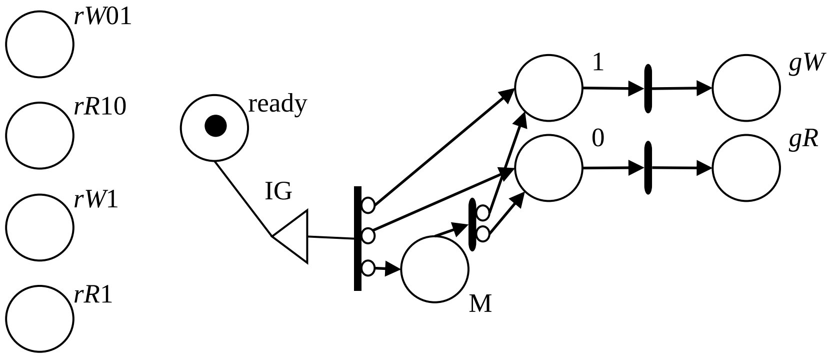

Fig. 6 Responding to requests

The model in Fig. 6 is based on the assumption that the internal MUTEX state of 1 corresponds to granting to writer and 0 corresponds to granting to reader, without losing generality. Once the internal state 0 or 1 is secured, the corresponding grant may take non-zero time to appear, represented by the timed transitions on the right hand side. If this degree of representation is not needed, these may be replaced with instantaneous transitions. The internal state M causes the entire process to delay with the only activity being the metastability settling. Note that this settling sub-net is less complex than Fig. 3 as in a MUTEX it is not possible for external influences to hard set M to one of the digital states. Because requesters would not withdraw outstanding requests, the only route out of the M state is via settlement.

The internal state is only represented between the start of a grant computation (the left hand transition firing) and the next grant commitment (one of the right hand transitions starting to fire). Although the actual circuit would maintain an internal state always, it is only of functional relevance in between those events. Hence the model is simplified this way.

The input gate IG has the following predicate and function:

Gate	**Predicate**	**Function**
IG	`Mark(ready)==1 && (Mark(rW1)==1 || Mark(rR1)==1)`	`Mark(ready)=0`

This means that grant computation starts when the place ‘ready’ is marked and at least one of the request signals is present (*rW*1 and/or *rR*1 marked). And starting a round of grant computation unmarks the place ‘ready’. Place ‘ready’ is marked again once a requesting process has finished using its granted access.

The probabilities in the grant computation starting transition need to reflect the following protocol:

- If neither *rR*1 nor *rR*01 is marked, i.e. the reader request is not present nor being issued, grant computation goes to the branch leading to internal place ‘1’ with a probability of 1. This leads towards granting writer access.
- If neither *rW*1 nor *rW*01 is marked, i.e. the writer request is not present nor being issued, grant computation goes to the branch leading to internal place ‘0’ with a probability of 1. This leads to granting reader access.
- If one of *rR*1/*rR*01 and one of *rW*1/*rW*01 are marked at the same time, i.e. both reader and writer requests are either present or being issued, grant computation goes to all three branches with appropriate probabilities, which may be obtained from experimental characterization of hardware.

Granting is represented by either place *gW* or place *gR*. The appropriate process taking its grant by taking the token from its corresponding grant place, and puts a token back to place ‘ready’ after the end of the access.

The normal assumptions of MUTEX arbitration apply. For instance, a requester is not supposed to withdrawn a request before a grant is issued in its favor. In order to represent all possible causes of metastability properly, request signals are modelled as taking non-zero time to set up (‘being issued’ is at least one distinct state). More precise modelling by having a multi-stage setup process for one or both of the request signals is possible with the method shown in Fig. 5, if necessary.

3 The ternary asynchronous variable

Larger control variables may be needed for more complex ACMs. Here we discuss the ternary control variable used in certain existing ACM algorithms, to show that larger control variables can be constructed out of Boolean asynchronous variables.

A ternary (base-3) variable taking three possible values (e.g., 0, 1, 2) may be implemented in a number of ways. The most straightforward is to use one-hot encoding and binary circuits. In other words, using three wires, with a maximum of one of then having a signal value of 1 at any time. Wire 0 being 1 and the other two wires being 0 indicate the value of the variable being 0, wire 1 being 1 and the other two wires being 0 indicate the value of the variable being 1, and wire 2 being 1 and the other two wires being 0 indicate the value of the variable being 2:

Signals on wires	**Variable value**
{1, 0, 0}	0
{0, 1, 0}	1
{0, 0, 1}	2
{0, 0, 0}	potential spacer (see below)
other values are not allowed	undefined

When such a variable is being sent from one independently timed process to another, the simplest way of avoiding confusion and reducing potential errors is to make use of spacers. In other words, when changing the value of such a variable, the current wire holding 1 is pulled down to 0 first, before another wire is set from 0 to 1. Briefly, the three wires hold the spacer signal {0, 0, 0}, which indicates the fact that the value of the ternary variable is being changed.

For instance, the following steps change the variable value from 0 to 2:

$\{1, 0, 0\} \rightarrow \{0, 0, 0\} \rightarrow \{0, 0, 1\}$

With such an implementation, or any other implementation using binary logic, the models derived in Section 2 can be directly used, as each ternary variable is implemented with binary signals.

It is worth noting that when reading the value of such a ternary variable, a spacer should cause a wait on the reader’s part as the value is not functionally valid. In other words, reading a ternary variable should be done using the following procedure:

```
wait until (v0=1 or v1=1 or v2=1);
read v;
```

In such a scheme, metastability can only happen when changing to spacer or changing out of spacer. The spacer scheme qualitatively trades spacer delay for additional variable value safety.

Ternary logic based on using three different analogue signal values to represent the three variable values is conceptually possible. It is worth noting that these circuits have been ignored in general by researchers in the field of asynchronous data communication. Modelling such implementations is out of the scope of this paper.

4 Case study

In this section we investigate the usage of the asynchronous variable models in a wider context, i.e. when these variables are used as control variables in ACMs. Here the examples used are the 'pool' or 'RR-OW' type, with a single logical buffer space, mechanisms that aim to provide the reading and writing processes with full timing independence. This means that overwriting of previously unread data stored in the buffer, and rereading of previously read data, must be allowed. This is intuitive if the buffer has a limited number of spaces (in this case a single space), and potentially allow multiple cycles of reader access in between two writer accesses and multiple writer accesses in between two reader accesses.

The fundamental assumption of these ACMs is that they serve as data connectors between the communicating processes between which a sequence of data items of the same type are transferred. In each cycle of writing and reading access, the relevant communicating process transfers one item of data to or from the ACM.

4.1 A two-slot ACM

The two-slot pool ACM, proposed by Simpson in [4], attempts to accommodate the full asynchrony between the reading and writing accesses by using two physical data memory spaces, each enough to contain one item of data. When an access happens, the other access should maximally be able to occupy one slot, hence the intuition is that whenever an access needs to happen it always has a slot to point to. Control variables are used to make sure that reading accesses 'chase' writing accesses – each reading access tries to read from the slot containing the newest completely written data item, as indicated by the writing access's previous round. How the writing access chooses its slot can be more interesting. A seemingly totally safe method is to always avoid the reading access by going to the other slot not currently being read. This however has been shown to create the possibility that the reading access will keep rereading the same item of data whilst the writing access keeps overwriting to the other slot, if the two accesses are matched in their speed. Simpson's method is to have the writing access point to alternating slots in successive rounds regardless of where the reading is happening. This has been shown to lead to clashes on the same slot by both processes and violate data coherence, under certain circumstances.

The algorithm for the two-slot mechanism can be written as follows:

Write access	**Read access**
`w=!(l);`	`r=l;`
`d[w]=input;`	`Output=d[r];`
`l=w;`	

In this algorithm the two data slots are arranged into an array *d*[0..1], whose access is managed through the shared control variable *l*, which is Boolean and indicates the last complete written slot. The writing access uses its private control variable *w* to choose the next slot to be written. The reading access uses its private control variable *r* to choose the next slot to be read.

In the writing access the sequence of actions are: choosing the slot not accessed by the previous writing access, writing to that slot, indicating that slot to be the last written one. It is then assumed that the writing process ends and the master process that contains the writing process will execute a sequence of actions which includes the preparation of the next writing access round. Then the writing access will start again from the first statement. This is assumed to form a forever loop as long as the ACM continues to be used.

The reading access has the following sequence of actions: choosing the slot to read from, read from that slot. After a round of reading access ends, the master process that includes the reading access is assumed to perform actions not related to accessing this ACM, including making use of the data just read. A forever loop situation similar to the writing side is also assumed.

The ACM accesses can then be viewed as procedures or functions called by their respective master processes in each cycle of action.

A single Boolean control variable is shared between the two access procedures, *l*, which is written by the writing access and read by the reading access. It is used for the writing side to indicate to the reading side, and to itself, the immediate previous completely written slot. The passing of this Boolean variable through the reader statement `r=l` can be modelled with the method described in Section 2.

Here we investigate the mode of operation where the two-slot ACM works if *l* is always correctly read. This is guaranteed if the duration between two writing accesses is longer than a reading access. If this is not the case, the two-slot ACM can violate data coherence even if *l* is always correctly read and there is not much point in analyzing what happens when, for instance, *r* becomes metastable.

As the value of *l* itself is changed entirely during a writer internal statement unrelated to the reader, it cannot be metastable. As a result the model for *r* and the synchronizer is simpler in methods (a) and (c) as there is no such thing as metastability propagation between the two sides. For method (b), the MUTEX protects variable *l*, and both `l=w;` and `r=l;` need to be preceded by requests to the MUTEX and followed by request resets. The writer example is as follows:

```
rW=1;
l=w;
rW=0;
```

4.2 Quantitative model explorations

Models in SANs were constructed within the environment of the Möbius tool [10] and quantitative explorations of the behavior of the two-slot ACM studied in the same environment. In this first explorative attempt, the following assumptions are made:

- The smallest step, that of assigning the value of a Boolean variable, is set to unit time, called τ – this usually corresponds with somewhere in the picosecond range in current CMOS technology given typical latches;
- Both reader and writer processes have deterministic delays in their statements, to emulate real-time programs whose timings have deterministic specifications;
- The reader process is started after a stochastic delay after the writer process, to emulate non-deterministic phase differences between the two processes, the resolution of this phase difference is 0.1τ so that processes can by desynchronized by less time than a full Boolean variable value change;
- Test cases with both `l=w;` and `r=l;` taking τ, with `l=w;` taking τ and `r=l;` taking 10τ, as well as with `l=w;` taking 10τ and `r=l;` taking τ are explored;
- Reading is assumed to take 10 times the time as a reader binary variable statement and writing is assumed to take 10 times the time as a writer binary variable statement;
- The time distance between two write accesses and that between two read accesses is assumed to take 1000 times the time as a binary variable statement.

Basically we cover the cases where the writer is 10 times faster than, the same speed as, and 10 times slower than the reader. The only non-deterministic delays in the study comes from metastability settlement. We also tried one case where the writer's speed is related to τ, and hence that of the reader, by a random non-integer value, but that did not show up any new results or trends.

We only explored methods (b) – MUTEX protection for *l* and (c) – unprotected fully asynchronous access to *l* by both sides.

The quantitative results we obtained, collated from a number of experiments, are listed as follows:

Method	**Data error min**	**Data error mean**	**Data error max**	**Delay min**	**Delay mean**	**Delay max**
(b)	0	0	0	0	0.013τ	∞[1]
(c)	0	0.025%	100%[2]	0	0	0

These explorations are based on realistic assumptions of very low probabilities of the onset of metastability even with overlapping accesses (e.g. 1%) and fast metastability settlement (e.g. the mean settlement time $1/\lambda = 0.1\tau$). Data error is measured by how many reads produced output values that have not been written, and 'delay' in the above results relate to additional delay either side has to suffer because the two state-

[1] These are theoretical values derived from [5].

ments `l=w;` and `r=l;` took longer than normal time due to the asynchrony. The results confirm the intuitive notion that the two methods trade delay with correctness as the MUTEX method protects the variable by paying potential non-deterministic delay as a price, and the fully asynchronous method guarantees full delay predictability by paying potential data corruption. Although these qualitative points can be derived from existing methods this work provides a systematic way of generating quantitative trade-off maps for designers.

5 Conclusions and future work

Quantitative models of asynchronous variables including metastability and its settlement are developed using the formalism stochastic activity networks (SANs) and their use initially demonstrated through a case study conducted using the Möbius tool. Whilst the actual numbers obtained from the work so far may not have any practical significance, the fact that they can be obtained using the methods provided represents a new development in the modelling of variables being passed between two non-fully synchronized processes.

The models can be extended and used on the entire class of existing ACM solutions and the hypothesis that the method may be used for developing new ACMs are promising topics of future work.

References

1. M. Krstic, E. Grass, F. Gürkaynak, P. Vivet, "Globally asynchronous, locally synchronous circuits: overview and outlook," in IEEE Design & Test of Computers, 24,(5), pp.430-441, 2007.
2. L. Benini, G. De Micheli, "Networks on chips: a new SoC paradigm," in Computer, 35,(1), pp. 70-78, 2002.
3. Lamport, L., 'On interprocess communication: Parts I and II', Distrib. Comput., 1,(2), pp.77-101 1986.
4. Simpson, R., "Four-slot fully asynchronous communication mechanism", IEE Proceedings, Pt. E, 137,(1), pp.17-30, 1990.
5. D. Kinnement, Synchronization and arbitration in digital systems, Wiley, 2007.
6. Kleeman, L., Cantoni, A., "On the unavoidability of metastable behavior in digital systems", IEEE Trans. Comput, 36,(1), pp.109-112, 1987.
7. Clark, I., Xia, F., Yakovlev, Y., Davies, A., 'Petri net models of latch metastability', Electronics Letters 34,(7), pp.635–636, 1998.
8. Xia, F, Yakovlev, A., Clark, I., Shang. D., 'Data communication in systems with heterogeneous timing', IEEE Micro, 22,(6), pp. 58-69, 2002.
9. Sanders, W. and Meyer J., 'Stochastic activity networks: formal definitions and concepts', LNCS 2090, pp.315-343, 2000.
10. The Möbius tool, available at: https://www.mobius.illinois.edu/.