Algorithms for Signal and Message Asynchronous Communication Mechanisms and their Analysis.

Article *in* Fundamenta Informaticae · April 2002

DOI: 10.1109/CSD.2001.981765 · Source: DBLP

2 authors:

Fei Xia
Newcastle University

Ian G. Clark
Newcastle University

Algorithms for Signal and Message Asynchronous Communication Mechanisms and Their Analysis

Fei Xia* Ian Clark†
*Department of Computing Science, University of Newcastle upon Tyne, NE1 7RU, UK.
fei.xia@ncl.ac.uk
†Department of Electronic Engineering, King's College London, WC2R 2LS, UK.
igclark@iee.org

Abstract

This paper presents new algorithms for the Signal and Message asynchronous data communication mechanisms (ACMs) and their modelling and analysis using Petri net techniques.

1 Introduction

In heterogeneously timed systems, data interfaces often need to be maintained between subsystems not belonging to the same timing domain. The minimal form of this problem is the unidirectional passing of data between two single-thread processes. When the two communicating processes are not synchronized, it is often necessary to pass the data through some intermediate data repository, usually in the form of shared memory. This is schematically shown in Figure 1.

Figure 1: Passing data via shared memory.

An asynchronous data communication mechanism (ACM) is a scheme which manages the transfer of data between two processes not necessarily synchronized for the purpose of data transfer. It is assumed that the data being passed consists of a stream of individual items of a given type. It is also assumed that the processes in question are single thread cycles, one providing and the other making use of a single item of data during each cycle. The data's provider is known as the "writer" and its user the "reader" of the ACM.

Many ACM protocols and systems for classifying these protocols have been proposed in the literature. One such classification system, briefly described below, for ACMs based on their asynchrony properties, can be found in [1].

Formally, an ACM has a capacity, a non-negative integer constant, which is the number of data items it contains. Each data item an ACM contains is either read or unread, at any time. The basic data state of an ACM consists of the number of unread data items it contains.

Write data accesses are divided into writing and overwriting. Read data accesses are divided into reading and re-reading. Writing increases the data state by 1 (one more unread item in the ACM) and reading decreases it by 1 (one less unread item in the ACM) while overwriting and re-reading do not modify the data state. Overwriting may occur, if permitted by the ACM protocol, only when the ACM's data state is equal to its capacity, i.e. when all items of data in it are unread. Re-reading may occur, if permitted by the ACM protocol, only when the ACM's data state is 0, i.e. when none of the items of data in it is unread.

ACMs are classified according to whether overwriting and re-reading are permitted. This scheme is shown in Table 1.

Table 1: ACM protocols classification based on overwriting and re-reading permission

	NRR	RR
NOW	Channel	Message
OW	Signal	Pool

In Table 1, NOW stands for no overwriting and NRR stands for no re-reading, etc. The names Channel, Message, Signal and Pool are given to ACMs demonstrating the respective asynchrony properties corresponding to their locations in the table. Some of these names have been retained from related literature [2].

An important issue with ACMs is the blocking/non-blocking characteristics by the data state of the access processes. For instance, the reader may be blocked from accessing the ACM if there is no unread data in it.

In terms of such blocking of data accessing by the data state, if re-reading is permitted (RR) there is no blocking of the reader and if overwriting is permitted (OW) there is no blocking of the writer. If re-reading is not permitted (NRR), the reader must wait when the data state is 0. If overwriting is not permitted (NOW), the writer must wait when the data state equals the ACM's capacity.

Most published ACMs have non-zero capacities. Special cases with zero capacities have also been mentioned [2].

Most computer systems, if they do not interface to analogue or real-time environments, will only use Channels for communication, in the form of FIFO/LIFO/RAM buffers. This is because for these systems, asynchrony is secondary to the preservation of data. These solutions are without data loss and repetition so have to maintain some synchronization. We do not cover these well-studied ACMs in this paper.

Pool ACMs have been extensively studied [3]~[7], [9], [11]. The attention of this paper will be focused on the other two ACM types.

High-level definitions of both Signal and Message, which describe the basic asynchrony and data passing requirements, will be given in Petri net [8] form. Algorithmic forms of these ACMs, derived from results of hardware synthesis in earlier work [1], are then described. Detailed Petri net models of these algorithms describing all important ACM properties will be developed. Then the algorithms will be analysed in detail to prove their properties.

2 Signal and Message ACMs

In order to describe the relations of asynchrony, data states, reading and writing in the context of ACMs, Petri net [8] models have been developed. These high-level models define ACM protocols in sufficient detail so that the asynchrony and data passing properties are clearly described.

For simplicity, all models in this paper assume that the relevant ACMs have data capacity 1. In other words, an ACM contains one item of data at all time. This does not represent a loss of generality. Models introduced here can be easily extended to cover non-unity capacity cases. Examples of these extended models can be found in [9].

2.1 Petri net definitions of Signal and Message

In complex systems, the temporal requirements of different processes may differ considerably. Some processes may exist in a very tight temporal regime for real-time and/or safety critical reasons. For such processes, it is important to be able to predict their progress with relation to time. This means that they are often mapped to hardware controlled by predictable, often rigid, clock signals. Other processes may exist in a fairly relaxed regime with respect to timing. For such processes concerns of maximizing efficiency, whether of power or of silicon, may outweigh predictability of their temporal behaviour with regard to real-time.

In many applications, there may be a need for processes existing in real-time and flexible time domains to communicate by passing data. This is where the Signal and Message type ACMs may be most relevant.

The basic definition of Signal treating read and write accesses as atomic is shown in Figure 2. Here re-reading is not allowed, normal writing occurs when the data state is 0 and overwriting occurs when the data state is 1.

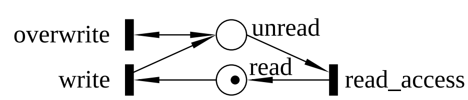

Figure 2: Signal with atomic reading and writing.

It is obvious from this definition that Signal is most useful in connecting a writer process which exists in a relatively tight temporal regime with a reader process which exists in a relatively flexible temporal regime. By disallowing re-reading, the reader side of the protocol is data-driven. By allowing overwriting, the writer side of the protocol is event-driven or time-driven. In other words, the writer enjoys temporal decoupling from the state of the ACM. Data passing through a Signal may be of the interrupt/exception type, where the generator of these requests must be accorded temporal independence, and a new request would supersede any previous requests not yet handled.

At the same basic level, the Message protocol is defined in Figure 3. This is a perfect mirror of the basic model of Signal, showing that Message is the dual of Signal. Message, therefore, will be most useful in connecting a more rigidly timed reader with a more flexibly timed writer.

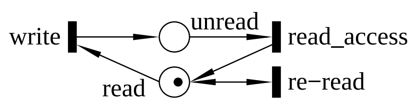

Figure 3: Message with atomic reading and writing.

A new "message" type data item would not be generated by the writer if previous ones have not been acted on. This is different from "signals" which are generated regardless of whether previous ones have been dealt with by the reader.

These models are sufficient to illustrate the concept behind the Message and Signal type ACMs which connect two sides with different priorities for asynchrony and data passing.

In practice, write and read data accesses are generally processes that take time. It is overly restrictive to specify that within an ACM, write and read accesses must be atomic relative to each other, as in the models of Figures 2 and 3. In real implementations, in the interest of enhancing asynchrony, it is often desirable to allow one side to be non-blocking to the other side, depending on the protocol.

For Signal, the data state may not block the writer. If this non-blocking is extended so that reader data accesses may not block the writer either, the writer can be made independent to both the Signal ACM and its reader. On the other hand, the data state may already block the reader, it makes sense to extend this blocking so that writer data accesses may also block the reader. This extension of blocking may reduce implementation size, and possibly enhance power efficiency. Such a definition of Signal, with full temporal independence for the writer and blocking of the reader by both data state and writer accesses, is shown in Figure 4.

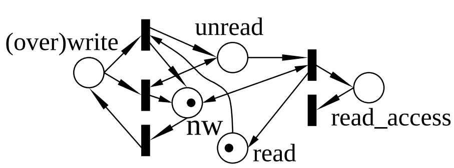

Figure 4: Signal with non-blocking read and blocking write and overwrite.

The corresponding definition of Message, with both read and write accesses treated as non-atomic and appropriate blocking characteristics, is shown in Figure 5.

In these models, place "nr" denotes the state of not reading

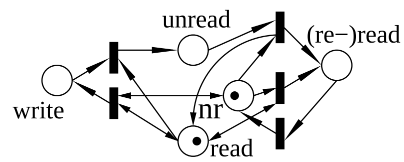

Figure 5: Message with non-blocking write and blocking read and re-read.

and not re-reading and place "nw" denotes not writing and not overwriting. Only when these places are marked and with the correct data states can the opposite accesses begin. Therefore these places specify the blocking by the more temporally independent access process of the opposite one.

2.2 Data properties of ACMs and slots

In addition to asynchrony, other important properties of ACMs exist, some having been investigated in previous work. The main data properties include data coherence, data freshness, data sequencing, data loss and data repetition.

Data loss and data repetition are the results of overwriting and re-reading, respectively, and are inevitable consequences of no blocking and bounded ACM implementations. Data loss occurs with overwriting, which causes one unread item of data in the ACM to be never available to the reader. Data repetition occurs with re-reading, with the reader obtaining an item of data it has already obtained in a previous cycle.

Data coherence refers to the integrity of individual data items being passed through an ACM. Items of data obtained by the reader should not have been modified since their introduction by the writer. In other words, data items passing through an ACM should retain their individual integrity or atomicity. If an item of data changes between the writer and the reader, data coherence is said to be lost.

Data freshness refers to the desire for the reader to obtain the most up to date data item in an ACM that satisfies the particular protocol. With a capacity 1 Signal or Message, any read access should obtain the data item introduced by the last write access completed before the read access starts.

Both data coherence and data freshness are specified in the models of Figures 4 and 5. Data coherence is implied by the non-breakability of individual tokens representing the number of read and unread data items. Data freshness is implied in the updating of the data state by both access processes.

In the terminology of multi-slot ACMs [3], a "data slot" is a unique portion of the shared memory which may contain one item of data. It is obvious that a Signal or Message in the form of Figures 4 and 5 cannot be implemented with only one data slot, since one slot cannot possibly support writing and reading at the same time and guarantee data coherence.

In ACM implementations with more than one slot, it is not possible to maintain strict data freshness and the kind of full temporal independence provided for the writer (reader) by the definition of Figure 4 (Figure 5) at the same time [3]. The accepted practice is to put priorities on the properties. The most natural system puts the asynchrony requirements first, data coherence second and data freshness third. In any system with data freshness requirements being relaxed, it is important to monitor the data sequencing behaviour.

Data sequencing is somewhat related to data freshness. It is maintained if the data items obtained by the reader has the same sequential order as that provided by the writer. If an ACM implementation is only able to maintain relaxed data freshness, it is preferable if it also maintains data sequencing. For ACMs that can maintain the strictly defined data freshness, data sequencing is obviously automatically maintained.

Previous work has indicated that it is desirable to minimize the number of slots in multi-slot ACM implementations [7]. The advantages include smaller hardware expenditure in both slot memory and control circuits, leading to better temporal performance and higher hardware reliability.

Most of the software solutions in the literature also spend considerable effort in the reduction of the number of slots needed for any particular ACM specification [3]~[6].

It is therefore important that solutions with just two data slots be found for both Message and Signal if possible.

3 Signal and Message with two slots

Circuits for 2-slot Signal and Message ACMs have been successfully synthesized in related work [1]. Based on these circuits and following the conventions introduced in [3], we have developed 2-slot Signal and Message algorithms.

Reader or writer must be directed to a correct slot before any data access. This must be accomplished through steering mechanism provided by the ACM. It is desirable to keep such a steering mechanism simple and its actions as near to atomic as possible. In digital systems, the actions nearest to atomic are the setting and resetting of a binary variable.

A 2-slot Signal algorithm is shown in Figure 6. In this algorithm, each of the reader and writer access procedures consists of consecutive statements which are assumed to be executed in a forever loop. The two data slots are labelled slot 0 and slot 1. Two binary $control\ variables$, w and r, are used to manage the steering of the writer and reader to the appropriate slots. An additional binary variable g is used as a “guard” to prevent read data accesses to start in the middle of a write data access. The writer process progresses through these steps: wg (guard up); wr (write data access); wg0 (guard down); w0 (writer slot steering); next wg; etc. The reader process has the following steps: r0 (reader slot steering); rd (read data access incorporating waiting).

writer	reader
wg: $g := 1$	r0: $r := \overline{r}$
wr: write slot w	rd: if* $w \neq r$ and $g = 0$ then
wg0: $g := 0$	read slot r
w0: $w := \overline{r}$	else wait

Figure 6: Algorithm of 2-slot Signal.

The guard g is 1 during any write data access (wr). By monitoring this variable the reader does not initiate a read data access (read slot r) when a wr statement is in progress.

For slot steering, statement w0 prepares the next write data access by setting the writer control variable w to the complement value of r, while statement r0 prepares the next read data access by setting the reader control variable r to its own complement. Thus w0 implements a measure of avoidance so that the writer tries not to access the same slot the reader may be accessing, thus improving data coherence. And r0 indicates the desire of avoiding re-reading and enhancing the probability of maintaining data freshness.

The data access statements (wr and rd) demonstrate the difference between the two sides. Since writing is always allowed, statement wr specifies writing to slot w in all cases, without regard to the data state or the situation of the reader. However, reading is only allowed when the data state is 1 and the writer is not in the middle of a wr statement, so statement rd is a choice of either waiting or reading from slot r depending on the current values of w and g.

At the beginning of statement rd, if $w \neq r$, the writer is either ready to access slot $\overline{r}$, or accessing it, or has just completed such an access but has not been able to indicate this completion to the reader by the next w0. This implies that slot r contains the latest data item in the ACM known to the reader, and should therefore be accessed provided the guard (g) condition is correct, statement r0 having made sure that slot r is not the one accessed by the previous reader cycle. On the other hand, if $w = r$ at the beginning of rd, the writer is either ready to access slot r, accessing it, or just completed such an access but has not been able to indicate this completion to the reader by the next w0. This implies that slot $\overline{r}$ contains the latest data item in the ACM known to the reader. Since statement r0 means that slot $\overline{r}$ was the slot accessed during the previous reader cycle, to avoid re-reading the reader must be made to wait, until w changes.

In hardware with self-timed circuits, such waiting can be made very energy efficient, the change of signal w providing the "trigger" to reactivate the reader part of the circuit [1]. In software implementations, such waiting may be implemented with polling or effective polling (de-scheduling and re-scheduling of the reader task in a multi-tasking element). In this case, the "if*" construct acts like "while" loops. However, by using "if*" instead of "while" here we want to emphasize the desire to move away from polling as much as possible. Upon termination of the wait, which is caused by a condition of the "if*" construct having changed, the reader continues rd, i.e., carries out the read slot r action, rather than jump to the next r0 without reading. This is true for all "if*" constructs in this paper.

writer	reader
wr: if* $w \neq r$ and $g = 0$ then	r0: $r := \overline{w}$
write slot w	rg: $g := 1$
else wait	rd: read slot r
w0: $w := \overline{w}$	rg0: $g := 0$

Figure 7: Algorithm of 2-slot Message.

The algorithm for a 2-slot Message is shown in Figure 7. This is the mirror image of the Signal algorithm of Figure 6. All arguments for the Signal algorithm can be reversed and applied to it.

In the interest of simplifying the mechanisms, we have tried to use the smallest number of binary variable assignment statements in building the steering and guard mechanisms. However, if the asynchrony requirements can be slightly relaxed, much simpler algorithms may be used to implement Signal and Message, by eliminating the guard mechanism. In the case of Signal, if the reader is allowed to start read data access in the middle of a write data access, provided not to the same slot, the simplified algorithm shown in Figure 8 may be adopted. Similarly, a dual algorithm for Message is shown in Figure 9.

None of these algorithms can be regarded as valid based on the reasoning presented in this section alone. They will be formally modelled and analysed in the following sections for the important asynchrony and data properties.

writer	reader
wr: write slot w	r0: $r := \overline{r}$
w0: $w := \overline{r}$	rd: if* $w \neq r$ then
	read slot r
	else wait

Figure 8: Simpler algorithm of 2-slot Signal.

writer	reader
wr: if* $w \neq r$ then	r0: $r := \overline{w}$
write slot w	rd: read slot r
else wait	
w0: $w := \overline{w}$	

Figure 9: Simpler algorithm of 2-slot Message.

4 Modelling and analysis

Petri net models of the Signal and Message algorithms proposed in Section 3 have been established for the study of all the important properties including the asynchrony requirements and data coherence, freshness, and sequencing. The ultimate goal of this study is to find out if the algorithms correctly implement the definitions given in Figures 4 and 5. These models are mainly in basic P/T net format with additional techniques employed to simplify the modelling.

4.1 Or-arcs

OR-arcs [11] are an extension to standard Petri nets for the purpose of reducing model size without affecting model functionality. In the construction of safe, cyclic Petri net models with reference arcs (arcs with arrows at both ends, simulating a pair of arcs pointing to opposite directions between the same pair of transition and place), multiple transitions are normally required to remove a token from places representing the previous value of a variable. Figure 15 contains an example of the use of or-arcs. A transition connected by or-arcs to multiple places takes a token from one and only one of the relevant places to fire (exclusive OR). Graphically or-arcs are shown as arcs with a dashed line rather than the standard solid line, and linked using another dashed line indicating the XOR set. Multiple sets of or-arcs can be used, which is when or-arcs provide the best model reduction. The use of or-arcs can help reduce human error in the construction of large, complex models. In the example in Figure 15, the two "end wr" transitions replace six transitions required in the equivalent traditional P/T net model. Or-arcs do not introduce any additional modelling power. They just make models more compact for easier generation and reading.

4.2 Monitoring subnets

Once the Signal and Message algorithms have been modelled in Petri nets, they need to be tested for the asynchrony and data properties mentioned earlier. This is performed by adding a monitoring subnet for each property individually. Each monitoring subnet connects to the main model via reference arcs, thus not affecting the operation of the main model. Since the properties being monitored are modified by each algorithm statement, the model of a statement needs to be in an atomic unit with its relevant part of the monitoring subnets. It is possible to achieve the effect of such atomicity by the use of an "enable" place: The model of each algorithm statement takes the token from place "enable" which is not placed back until the monitoring actions related to this statement have been completed.

Coherence is maintained if the reader never obtains a data item which contains parts from different items provided by the writer. The reader and writer never accessing the same slot at the same time is a sufficient condition for coherence.

Reading from and writing to a particular slot are states explicitly represented by places in the main algorithm model. The subnet monitoring coherence is shown in Figure 10. If both the reading from slot *a* and the writing to slot *a* places are marked then the transition is enabled which when fired puts a token in the "coherence lost" place. All potential failure modes for data coherence can then be found through reachability analysis. The operation of such a monitoring subnet, because of the reference arcs, does not affect the running of the main model net in reachability analysis.

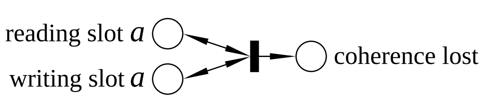

Figure 10: Monitoring data coherence

Data freshness refers to the desire that the reader does not obtain stale data. Strict data freshness is maintained only if at the beginning of every read data access (rd), slot r contains an item of data which is the last item to be completely written and indicated by the writer process. The test for this is described in Figure 11. Figure 11 contains two independent time lines, one each for the writer (top line) and reader (bottom line). For the writer time line, the duration labelled "wr" is the time taken by statement wr and the duration labelled "w0" is the time taken by statement w0. The arrow pointing to the end of the "wr" duration means that at the end of wr, the slot just written into is set to "fresh". "Fresh", together with "valid" in Figures 12 and 13, are monitoring states set and reset for each slot by the freshness monitoring net. Figure 11 shows the case for a 2-slot algorithm where when one slot becomes fresh the other one automatically becomes not fresh. For algorithms with more than 2 slots all other slots would be set to not fresh.

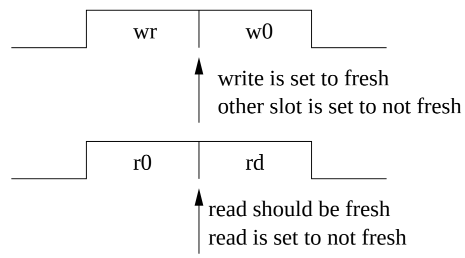

Figure 11: Checking strict data freshness for Signal.

For the reader time-line the arrow points to the beginning of rd, indicating that strict data freshness is maintained if at the beginning of an rd the slot to be accessed is fresh.

This kind of strict freshness is, however, not possible to achieve if the statements r0, rd, wr and w0 can interleave with each other, i.e. a possible sequence of execution could be wr, r0, w0, rd. Realistically, in the convention of [3], the data freshness requirement needs to be relaxed when there is an overlap between the writer post sequence (w0) and the reader pre-sequence (r0). In other words, since the reader selects a slot to access with r0 and the writer indicates its slot selection with w0, strict data freshness should not be required when these statements do not form atomic groups with their respective data access statements. This relaxed data freshness is shown in Figure 12 and is in the convention of [3]. This form of data freshness was also discussed in depth in [7] and [11]. Figure 12 shows the addition of the arrow at the start of the read pre-sequence which sets to not valid for all slots except the slot which is fresh, i.e. the last one written. As long as the reader obtains the data from this slot or a newer item, relaxed freshness is maintained. The additional state valid is needed to keep track of relaxed data freshness.

The subnets modelling the events at the arrows need to be made atomic with the models of their related statements, requiring the "enable" place as described above.

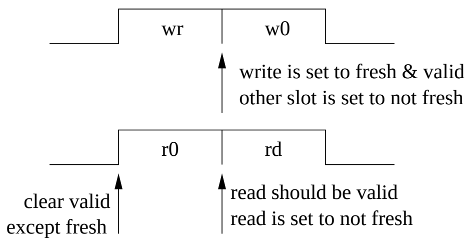

Figure 12: Checking relaxed data freshness for Signal.

The scheme for testing the data freshness property of the Message is shown in Figure 13. Monitoring subnets have also been constructed for data sequencing etc.

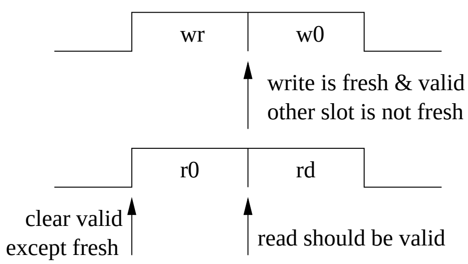

Figure 13: Checking relaxed data freshness for Message.

4.3 Modelling Signal algorithm statements

In Figure 14, the main model of the read and write data accesses of slot 0 of the 2-slot Signal algorithm is shown. The monitoring subnet for re-reading is also included in this diagram. Places "wr", "w0", "rd" and "r0" denote the "ready" state for the relevant statements. Transitions with "strt" and "end" in their names represent the start and end of the relevant statements respectively. The place "re-read0" becomes marked if re-reading happens on slot 0, which is how reachability analysis can show up possible faults in the design of the algorithm with regard to the asynchrony requirement of no re-reading.

The top "strtrd" transition in Figure 14 is provided to monitor the cases of potential start reading slot 0 when the writer is in the middle of writing slot 0. This transition is needed because in this situation the main model of the slot shows a data state of neither read nor unread because of the way the writer is modelled. The middle "strtrd" transition in Figure

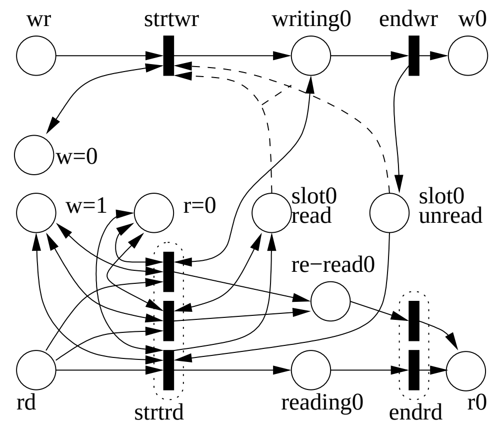

Figure 14: Model of one of the slots of the 2-slot Signal.

14 is provided to monitor the cases of potential start reading slot 0 when it has a data state of read. Since all asynchrony and data properties are supposed to be maintained by the control variables, these transitions must be provided to monitor the potential of control variables not working as designed. In other words, if either place "slot0 read" or place "writing0" is ever simultaneously marked with "r=0" and "w=1", these monitoring subnets will catch it in reachability analysis. If, on the other hand, the algorithm conforms with the specification, these transitions should never fire in reachability analysis.

The model for read/write accesses of slot 1 is exactly the same shape as Figure 14, with the appropriate label changes. The models for setting and resetting binary control variables (statements r0 and w0) follow the Petri net modelling conventions of such actions and are not shown here [9], [11].

4.4 Modelling Message algorithm statements

The main model of the read and write data access of slot 0 of the 2-slot Message is shown in Figure 15. The monitoring subnet for overwriting is included in the diagram. Place "ovwr0" becomes marked if overwriting occurs on slot 0. If it is ever marked in reachability analysis the algorithm has a fault in its design with regard to the asynchrony requirement of no overwriting.

The same concerns raised in 4.3 have been catered for here with similar subnets. The "endwr" transitions in Figure 15 are slightly different from the "endrd" ones in Figure 14 because the local data state of read/unread is updated at the start of a read access and end of a write access.

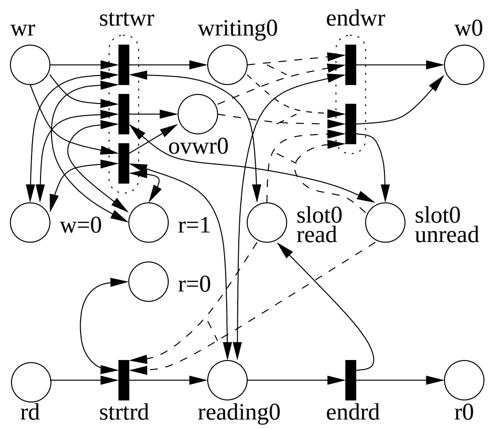

Figure 15: Model of one of the slots of the 2-slot Message.

4.5 Modelling the reader and writer cycles

Each of the mechanism algorithms consists of two cyclic sequential processes connected with each other only through the setting/resetting and testing of control variables as well as the possible waiting of the less temporally independent side. With the statements modelled with "ready" places followed by a statement subnet which can be treated as a virtual transition at a higher level, an entire mechanism can be represented by these subnets assembled into two state machines. This scheme is shown in Figure 16 for the Signal algorithm. The model for the Message algorithm has a similar structure.

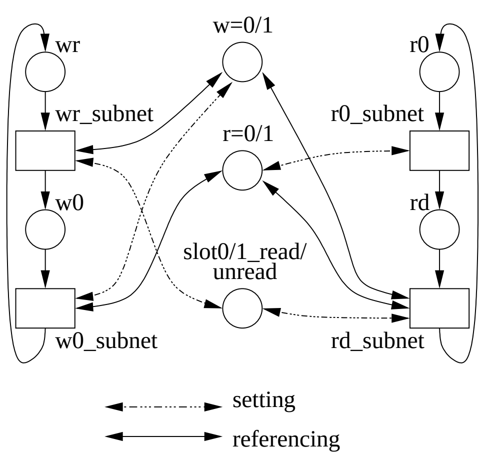

Figure 16: Overall model structure for the Signal algorithm.

4.6 Analysis results

The overall models of both the original algorithms (Figures 6 and 7) and the simpler algorithms (Figures 8 and 9) are used in reachability analysis with the monitoring nets to catch any "bad states" for violations of the required asynchrony and data properties. The results from this analysis show that neither of the original algorithms violates the asynchrony requirements. In other words, the Signal algorithm supports write data access at all times but causes read data accesses to wait when the writer is accessing and when there is no unread data in the ACM, while the Message algorithm supports read data access at all times and causes the write data accesses to wait under the prescribed conditions.

For both original algorithms, there is no violation of data coherence. In other words, reading and writing the same slot at the same time never happens.

For both original algorithms, there is no violation of data freshness for the relaxed definition of freshness introduced earlier. However, as expected, neither original algorithm can maintain freshness in the strict sense.

Because strict freshness is not maintained by either original algorithm, data sequencing was tested. It was found that both original algorithms maintain data sequencing.

For the simpler algorithms, it was found that they do not strictly observe the asynchrony requirements, in the sense that the process with the weaker temporal independence is allowed to start data accesses while the other side is in the middle of a data access under certain conditions. For Signal this is a partial violation of the definition in Figure 4, which specifies that write accesses block read accesses. The same can be said, in reverse, for Message. However, these algorithms do maintain data coherence, indicating that as expected, these accesses are directed to different slots.

Similar to the original algorithms, the simpler ones do not maintain strict data freshness, but do observe the relaxed data freshness requirements shown in Figures 12 and 13.

Undesirable operating modes of the algorithms have been found during the analysis by studying the state trajectories from the reachability analysis. These relate to the inability to maintain strict data freshness. For the Signal algorithm, if the distance in time between wr and the next w0 statements is large relative to the speed of the reader, long waits may be forced on the reader. If the distance in time between r0 and the next rd statements is large relative to the speed of the writer (for instance, if many complete cycles of the writer fit into such a gap), very old data may be read. Reading such old data, although not violating the more relaxed data freshness requirement, can be very bad in practice.

Since there is no assumption about the relative speeds of the reader and writer processes, these undesirable operating modes are a real concern. By ordering the algorithms in the way of Figures 6, 7, 8 and 9, the desire to avoid such modes as much as possible is made clear.

The algorithms have also been found to be slightly sensitive to initial conditions. When the writer and reader processes are assumed to start from the state with their respective first statements being "ready", if the initial values of variables w and r are the same, there is the possibility of the unnecessary loss of data for Signal and the unnecessary repetition of data for Message. Such unnecessary loss or repetition happens only once and after the ACMs settle into steady state operation (after both reader and writer have performed at least one entire cycle) they will never recur. Nor will they happen if w and r are initialized with different values.

All the above analysis results are from models which assume that the control variable statements w0 and r0 are atomic with respect to the opposite side. In other words, an r0 cannot be interleaved into by any writer statement and a w0 cannot be interleaved into by any reader statement. With proper protection of these control variables, which only incurs a very insignificant penalty in asynchrony, this can be accomplished at the level of hardware [1], [7].

If these control variable statements cannot be regarded as atomic, however, the simpler algorithms would fail to maintain data coherence and data freshness in certain situations. The original algorithms, however, do maintain data coherence and the relaxed data freshness under such non-atomic assumptions.

5 Three-slot Signal algorithm

The two slot Signal algorithms, when operating in some of the undesirable modes mentioned in 4.6, has an effective data capacity of 0 rather than 1. This happens when there have been repeated overwrites on a slot while the reader is occupying the other one. If the reader finishes the data access and then comes back for another rd, and if now the writer happens to be in the middle of a wr, the reader is forced to wait for the current item being delivered by this wr, because those delivered by previous wr's have been overwritten and lost. For a strictly capacity 1 ACM, in this situation the reader should be allowed to pick up the item delivered by the previous wr (the current wr, not having completed, should be unknown to the reader and therefore the data item from the previous wr must be viewed as the unread item if it has not been read before). However, this is not physically possible with only two data slots.

An algorithm using three data slots and ternary control variables have been developed based on experiences gained through the development of the two slot ones. This algorithm is shown in Figure 17.

writer	reader
wr: write slot w	r0: if* $r \neq l$ then
w0: $l := w$	$r := l$
wg: $g := 1$	else wait
w1: $w := \neg(l, r)$	rd: if* $w \neq r$ and $g = 0$ then
wg0: $g := 0$	read slot r
	else wait

Figure 17: Algorithm of 3-slot Signal.

Using the same techniques introduced earlier, this algorithm has been modelled and analysed. The analysis proved that it maintains all the required properties, except strict data freshness, under all modes of asynchrony. This is also true when the statements are regarded as non-atomic. Significantly, this algorithm behaves as a capacity 1 ACM in all its operating modes.

6 Summary and future work

A new classification system for ACMs was introduced. Petri net definitions of Signal and Message ACMs were described. Algorithms for 2-slot Signal and Message ACMs were designed. These algorithms were then proved with the help of Petri net models to uphold all of the property requirements from the mechanism definitions except for strict data freshness and in the case of the simpler algorithms, certain types of waiting.

The analysis results for the simpler algorithms can be applied directly to the hardware solutions described in [1] because these algorithms are a direct translation of the hardware implementations. In these hardware implementations the control variables w and r are protected by arbiter-like elements, making their setting and reading atomic processes. This means that the hardware solutions described in [1] successfully maintains coherence and relaxed freshness.

The two slot Signal algorithms have been found to behave as a capacity 0 ACM under certain operating modes. To obtain a pure form of capacity 1 Signal, a three-slot algorithm has been developed. It has been proved to maintain the important properties by using the same Petri net based modelling and analysis techniques.

More work is needed to obtain a more complete understanding of the relationships between the various types of ACMs and how they may be combined to form new communication links for specific requirements. Quantitative studies of Signal and Message ACMs, similar to those done for Pool ACMs in [12], may be carried out as part of this work to help build a clearer picture of how ACMs work and how to make the best use of them.

There is also scope for further investigation in how to implement ACMs using multiple slot techniques with capacities other than 1 and between more than one reader/writer pairs. A process of synthesizing ACMs from requirements, rather than the design and verify approach presented here, is also being developed.

7 Acknowledgements

This work was supported by the EPSRC through grants GR/L93775 at Newcastle University and GR/L92471 at King's College London (Project COMFORT). The authors wish also to thank A. Yakovlev, H. Simpson, D. Shang and A. Bystrov for enlightening discussions.

References

[1] Yakovlev, A., Xia, F. and Shang, D., "Synthesis of a signal-type asynchronous data communication mechanism and its hardware implementation", Async2001, March 2001, Salt Lake City, Utah, USA.

[2] Simpson, H.R., Campbell, E., "Real-time network architecture: principles and practice", Proc. AINT'2000, Asynchronous Interfaces: Tools, Techniques and Implementations, p.5 and handouts, TU Delft, The Netherlands, July 19-20, 2000.

[3] Simpson, H.R., "Four-slot fully asynchronous communication mechanism", IEE Proc. Vol. 137, Pt.E, No.1, PP.17-30, January 1990.

[4] Tromp, J., "How to construct an atomic variable", Proc. 3rd Int. Workshop on Distributed Algorithms, Nice, LNCS, Spring Verlag, pp.292-302, 1989.

[5] Kirousis, L.M., "Atomic multireader register", Proc. 2nd Int. Workshop on Distributed Computing, Amsterdam, LNCS-312, pp.278-296, Springer Verlag, 1987.

[6] Chen, J. and Burns, A., "Asynchronous data sharing in multiprocessor real-time systems using process consensus", Proc. 10th Euromicro Workshop on Real-Time Systems, pp.2-9, Berlin, Germany, IEEE Comp. Soc., June 17-19, 1998.

[7] Xia, F., Yakovlev, A., Shang, D., Bystrov, A., Koelmans, A. and Kinniment, D.J., "Asynchronous communication mechanisms using self-timed circuits", Proc. Async2000, 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems, Eilat, Israel, April 4-6, 2000.

[8] Peterson, J.L., Petri net theory and the modelling of systems, Prentice-Hall, 1981.

[9] Xia, F., "Supporting the MASCOT method with Petri net techniques for real-time systems development", Ph.D. Thesis, London University, King's College, January 2000. (Downloadable from: http://www.eee.kcl.ac.uk/~comfort/)

[10] Clark, I., Xia, F., Yakovlev, A. and Davies, A.C., "Petri net models of latch metastability", Electronics letters, Vol.34, No.7, pp.635-636, April 2, 1998.

[11] Clark, I.G., "A unified approach to the study of asynchronous communication mechanisms in real time systems", Ph.D. Thesis, London University, King's College, May 2000. (Downloadable from: http://www.eee.kcl.ac.uk/~ianc/)

[12] A. Madalinski, F. Xia, A. Yakovlev. "Studying the data loss and data re-reading behaviour of a four-slot asynchronous communication mechanism using stochastic Petri net techniques", 7th UK Asynchronous Forum, 20-21st December 1999, University of Newcastle upon Tyne.