School of Electrical, Electronic & Computer Engineering

Newcastle University

Modeling and Verifying Asynchronous Communication Mechanisms using Coloured Petri Nets

Kyller Gorgônio and Fei Xia

Technical Report Series

NCL-EECE-MSD-TR-2008-127

March 2008

Contact:

kyller@dee.ufcg.edu.br

fei.xia@newcastle.ac.uk

Supported by: EPSRC grant EP/C512812; CICYT TIN2004-07925, a Distinction for Research from the Generalitat de Catalunya; Nokia Institute of Technology/Nokia do Brasil through cooperation project with Federal University of Campina Grande

NCL-EECE-MSD-TR-2008-127

School of Electrical, Electronic & Computer Engineering,
Merz Court,
University of Newcastle upon Tyne,
Newcastle upon Tyne, NE1 7RU, UK

http://async.org.uk/

Modeling and Verifying Asynchronous Communication Mechanisms using Coloured Petri Nets

Kyller Gorgônio and Fei Xia

March 2008

Abstract

Asynchronous data communication mechanisms (ACMs) have been extensively studied as data connectors between independently timed concurrent processes. In previous work, two automatic ACM synthesis methods have been proposed. However, problems remain unresolved with the most asynchronous type of ACMs, the overwriting and re-reading bounded buffer (OWRRBB), especially with buffer sizes greater than one. In this work, a method of systematic modeling and verification of multi-cell OWRRBBs is presented. This method supports the study of these kinds of ACMs with regard to vital data and temporal characteristics.

1 Introduction

Allowing as much asynchrony as possible is one of the most important goals when designing communication schemes between asynchronous processes. And this task becomes more important when the size of computation networks becomes large and the traffic between the processing elements increases. One of the most important motivations for allowing maximal asynchrony in data communications is real time applications. Hard real time computational nodes need to have independent temporal motive powers (clocks) and the timing of thread executions should not be influenced by external data communications.

An *Asynchronous Communication Mechanism* (ACM) is a scheme designed to manage the transfer of data between a *producer* and a *consumer* processes that are not necessarily synchronized for data transfer. In the ACM model there is a shared memory to hold the data being transferred and some control variables. This scheme is shown in Figure 1. In this work the data consists of a stream of items of the same type, the writer and reader processes are single-threaded loops, and at each iteration a single data item is transferred to or from the ACM.

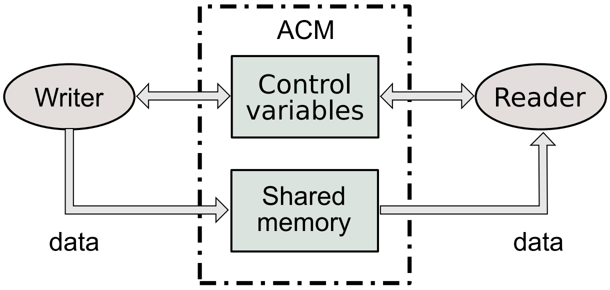

Figure 1: ACM with shared memory and control variables

ACMs may be of arbitrary size and the shared memory is organized as a ring of memory cells, as illustrated by Figure 2, each one being able to hold one data item. Each process attempting to access a specific cell has the access to it granted or denied according to the values of the control variables and the requirements the ACM should satisfy.

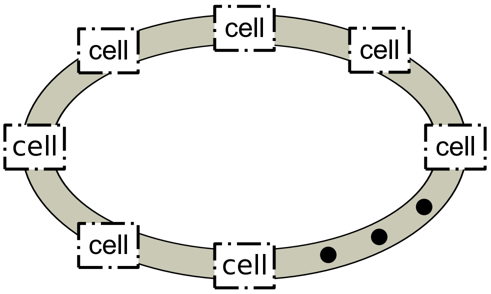

Figure 2: Multi-cell ACM scheme

Classical semaphores can be configured to preserve the coherence of write and read operations. However, this approach is not satisfactory when data items are large and a minimum locking between the writer and the reader is expected [Lam86]. By using single-bit unidirectional variables, the synchronization control can be reduced to atomic actions consisting of reading and writing them [Sim03], providing the safest solution for a maximum asynchrony between the processes. Variables are said to be *unidirectional* when they can only be modified by one of the processes.

ACMs are classified according to whether *overwriting* and *re-reading* are allowed or not [Sim03, XHC$^+$04]. Overwriting occurs when the ACM is full of non-read data, and in this case the producer may overwrite the data item in a cell causing the overwritten item to be lost. Re-reading occurs when all data in the ACM has been read before, and in this case the consumer is allowed to re-read an existing item. Table 1 shows such a classification. **BB** stands for a bounded buffer that does not allow neither overwriting nor re-reading. **RRBB** stands for an ACM that only allows re-reading. On the other hand, the **OWBB** scheme allows only overwriting. Finally, the **OWRRBB** scheme allows both re-reading and overwriting.

	No re-reading	**Re-reading**
No overwriting	BB	RRBB
Overwriting	OWBB	OWRRBB

Table 1: Classification of ACMs

For the re-reading ACM class, it is more convenient to re-read the item from the previous cycle rather than an item from several cycles before. For overwriting, the typical cases consist of overwriting either the newest or the oldest item [Fas01, GCX07, Sim03, YKXK98]. Overwriting the newest item [YKXK98] attempts to provide the best continuity of data items. Continuity is one of the primary reasons for having a buffer of significant size. Overwriting the oldest item is based on the assumption that newer data is always more relevant than older. The choice of a particular class of ACM for a certain job generally depends on data requirements and system timing restrictions [Lam86, Sim03]. This is generally true for real-time systems.

An introductory example

Now consider a simple ACM with three data cells. The single-bit (boolean) control variables r_i and w_i, with $i \in \{1, 2, 3\}$, are used to indicate which cell each process must access. Initially the reader is pointing at cell 0, $r_0 = 1$ and $r_1 = r_2 = 0$, and the writer at cell 1, $w_1 = 1$ and $w_0 = w_2 = 0$. The shared memory is initialized with some data. This scheme is shown in Figure 3.

The writer always stores some data into the ACM and then attempts to advance to the next cell releasing the new data. A possible trace for the writer is $\langle wr_1 wr_2 wr_0 wr_1 \rangle$, where wr_i denotes "*write data on cell i*", and a possible trace for the reader is $\langle rd_0 rd_1 rd_1 rd_2 \rangle$. Depending on how these traces interleave, coherence and freshness properties must be satisfied.

Coherence is related to mutual exclusion between the writer and the reader. For example, a possible

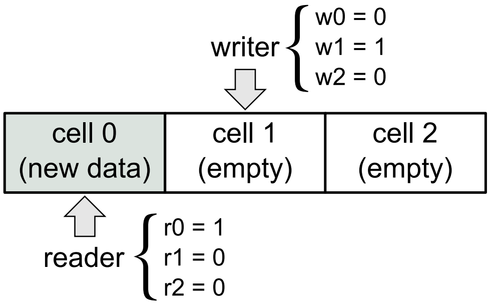

Figure 3: Execution of RRBB ACM with 3 cells

interleaving is $\langle wr_1 wr_2 rd_0 \cdots \rangle$. After the writer executing twice, the next possible action for both processes is to access cell 0. This potentially introduces a data coherence problem.

Freshness is related to the fact that the last data produced by the writer must be available for the reader. The reader always attempts to retrieve the oldest non-read data. This means that freshness imposes a specific data sequencing, i.e. the data is read in the same order that it is written. Data can be lost or read more than once, but the order is maintained. For the example above, one possible trace is $\langle wr_1 rd_0 wr_2 rd_1 rd_1 \cdots \rangle$. At the moment the reader executes the first rd_1 action, the writer has already executed a wr_2. This means that there is some new data on cell 2. But the reader is engaged to execute rd_1 again, violating freshness. For different ACM types there may be different notions of freshness. For instance, freshness may imply that the newest non-read data needs to be retrieved. In this paper we concentrate on the oldest available data paradigm.

For the example above, a correct trace is $\langle wr_1 rd_0 rd_1 wr_2 rd_1 wr_0 rd_2 wr_1 \rangle$. The sub-trace $rd_1 wr_2 rd_1$ does not contradict the fact that the reader only re-reads any data if there is no new one available. This is because after the first rd_1 there is no new data, then the reader prepares to re-read and from this point it will engage in a re-reading regardless the actions of the writer.

Observe that in the example above each process consists of an infinite loop, and at each ACM operation:

- The writer first writes to the shared memory and then tries to advance to the next cell by modifying its control variable w, if this is contradictory to the current values of the reader's control variable r, the writer waits. While the writer waits, the data item just written into the ACM is not available for the reader to read because the writer has not yet completed its move to the next cell.
- The reader first tries to advance to the next cell by modifying its control variable r, if this is contradictory to the current values of the writer's control variable w, no modification to r occurs. In either case the reader then reads (or re-reads) from cell r. Cell r cannot be accessed by the writer, even if its content has already been read by the reader.

In other words, at any time, each of the writer and reader processes "owns" a cell, and for data coherence purposes any cell can only "belong to" one of these processes at any time. Furthermore, since only binary control variables are used, the size of this description grows with the size of the ACM. This means that more variables are needed, and for overwriting ACM classes it is more difficult to correctly deal with all of them. Overwriting ACMs are also more relevant to real time applications because overwriting breaks any timing dependency of the reader on the writer. In [GCXY07] an automatic method for the synthesis of ACMs is proposed. It requires the generation of the complete state space of the ACM by exploring all possible interleavings between the reader and the writer actions. The state space of the ACM was generated from its functional specification then a Petri net [Mur89] model was obtained using the concept of ACM regions, a refined version of the conventional regions. Another approach for generating ACMs is introduced in [GCX07]. It is based on the synthesis of a Petri net model using a modular approach that does not require the explicit enumeration of the state space. The Petri net model is build by connecting a set of Petri net modules. Finally the model is translated to C++ source code.

These synthesis methods, however, have shortcomings. For instance, the low-level state space can become very large when both overwriting and re-reading are allowed, especially if the number of cells in the ACM is also large. This may render the ACM region based method very difficult or even impossible to use. The modular approach, on the other hand, is heavily based on heuristics and human experience, lacking a formal proof of its correctness.

This method requires the knowledge of "expected behaviors" of an ACM or its parts to construct the modules used in the assembly. Questions might be raised on the validity and rigorousness of the results obtained if verification cannot be satisfactorily carried out. For instance, valid questions may be raised on whether something believed to be an expected behavior at some low level detail might actually be in conflict with some specified overall ACM property such as data coherence or freshness.

In this paper, we try to study the expected behaviors of OWRRBBs of finite, but arbitrary, size. Such expected behaviors are at a detailed level drawn from previous experiences obtained from the successful synthesis of OWRRBBs of size one and other types of ACMs of arbitrary sizes. If such behaviors do indeed satisfy the important overall properties required of this kind of OWRRBBs, attempts at synthesizing using the modular method can then be carried out with high confidence, avoiding traversing the whole low-level state space. In effect, we will attempt to devise a method with which models can be obtained and verifications can be carried out at a much higher level of abstraction which will nevertheless provide for the same degree of confidence in future synthesis as complete low-level state space analysis may provide. And the method will include progressively more algorithmic models which hopefully will lead to better synthesis.

In the rest of the paper the set of behaviors expected from OWRRBBs of arbitrary sizes is formally described then verified for data coherence and data freshness. An ACM with this behavior set is then modeled using Coloured Petri Nets [Jen92, Jen97]. Simulating the CPN models produced Message Sequence Charts (MSC) [HT03] further clarifying the data and temporal characteristics of the set of behaviors, confirming that the CPN models are correctly derived. Finally, by specifying data coherence and data freshness using CTL, we are able to verify for these required OWRRBB properties on the CPN models. This is done with the ASKCTL model checker. The use of CPN to model the behavior of ACMs introduces the advantage of not requiring the designer to specify all details about the target policy. In this way, it is possible to introduce new policies without the need to construct its complete state space or to provide a low-level algorithmic description. This is done at a higher level of abstraction, allowing for simulation and verification with less effort during design. In this sense, the use of MSCs allows the validation of the model with respect to the expected traces of the ACM, which should be specified by a transition rule system.

2 Overwriting ACMs

The expected behavior of an overwriting ACM is described as a transition system. Each $\sigma = a_0 a_1 \cdots a_{j-1} a_j$, where $j < n$ and n is size of the ACM, defines the state of the ACM based on the data items available for reading. a_j is the last written data, and a_0 is the next data to be retrieved by the reader. The size of the ACM is given by its number of cells, i.e. the maximum number of data items the ACM can store at a certain time.

σ also express if the processes are accessing the ACM or not. This is denoted by flags in the a_0 and a_j items. a_j^w indicates that the writer is producing data a_j, and this data is not yet available for reading. Similarly, a_0^r is used to indicate that the reader is consuming data a_0. There are four events that change the state of the ACM:

- $rd_b(a)$: reading data item a begins.

- $rd_e(a)$: reading data item a ends.
- $wr_b(a)$: writing data item a begins.
- $wr_e(a)$: writing data item a ends.

The notation $\langle\sigma_i\rangle \xrightarrow{e} \langle\sigma_j\rangle$ denotes the occurrence of event e from state $\langle\sigma_i\rangle$ to state $\langle\sigma_j\rangle$.

The writer can add data in the ACM until it is full. In such case, the oldest data item is replaced and the writer proceeds its normal operation. The reader always tries to retrieve the oldest non-read data and, if all data in the ACM has been read before, then it attempts to re-read the last retrieved data item. Note that both processes are required not to wait when starting an access to the ACM.

The above behavior set is formally introduced by Definition 1. Rules 1-4 model the behavior set of the writer. Rules 5-8 model the behavior set of the reader.

Definition 1 (OWRRBB transition rules) *The behavior set of an OWRRBB ACM is defined by the following set of transitions (n is the number of cells of the ACM and the cells are numbered from* 0 *to* $n-1$*):*

1. $\langle\sigma\rangle \xrightarrow{wr_b(a)} \langle\sigma a^w\rangle \quad \textbf{if} \;\; |\sigma| < n$
2. $\langle a\sigma\rangle \xrightarrow{wr_b(b)} \langle\sigma b^w\rangle \quad \textbf{if} \quad |a\sigma| = n$
3. $\langle a^r b\sigma\rangle \xrightarrow{wr_b(c)} \langle a^r\sigma c^w\rangle \quad \textbf{if} \quad |ab\sigma| = n$
4. $\langle\sigma a^w\rangle \xrightarrow{wr_e(a)} \langle\sigma a\rangle$
5. $\langle a\sigma\rangle \xrightarrow{rd_b(a)} \langle a^r\sigma\rangle$
6. $\langle a^r\sigma\rangle \xrightarrow{rd_e(a)} \langle\sigma\rangle \quad \textbf{if} \;\; |\sigma| > 0 \;\wedge\; \sigma \neq b^w$
7. $\langle a^r\rangle \xrightarrow{rd_e(a)} \langle a\rangle$
8. $\langle a^r b^w\rangle \xrightarrow{rd_e(a)} \langle ab^w\rangle$

Observe that in state $\langle a^r b^w\rangle$ the next element to be retrieved depends on the order that events $wr_e(b)$ and $rd_e(a)$ occur. If the writer delivers b before the reader finishes retrieving a, then b will be the next data to be read. Otherwise, the reader will prepare to re-read a. It is also important to note that when the ACM is full of data and writer is starting a new access action, some data is lost. If the reader is accessing the ACM, with $\langle a^{rd} b\sigma\rangle$ in the data queue, then the second item in queue is overwritten. Otherwise, if the reader is idle, the queue contains $\langle a\sigma\rangle$ and first item is replaced.

Definition 1 was modeled using the Cadence SMV model checker and freshness and coherence properties were verified. Each process was modeled as an SMV module. In the SMV language, a module is a set of definitions, such as type declarations and assignments, that can be reused. Specifically, each process consists of a `case` statement in which each condition corresponds to a rule in Definition 1. The SMV model obtained from Definition 1 will be used in Section 3 to verify a lower level specification of the ACM. Next, the specification of the coherence and freshness properties is discussed. Previously [GCX07] the RRBB policy was modeled in SMV and verified.

Coherence

Verifying coherence requires to show that there is no reachable state in the system in which both processes are addressing the same segment of the shared memory. According to Definition 1, the reader always addresses the data stored in the head of σ, while the writer always addresses the end of the tail of σ. Verifying coherence in this model only requires to prove that every time the reader is accessing the ACM:

1. It is addressing the first data item, and

2. If the writer is also accessing the ACM, then it is not writing in the first location.

In other words, if at a certain time the shared memory contains a sequence of data $\sigma = a_0 a_1 \cdots a_{j-1} a_j$, with $j < n$, where n is the size of the ACM. Then, the following CTL formula should be satisfied:

$$\texttt{AG}\ (a^r \in \sigma \rightarrow (a^r = a_0 \wedge (a^w \in \sigma \rightarrow a^w = a_j | j > 0)))$$

Freshness

As discussed before, freshness is related to sequencing of data. Let us assume that at a certain time the shared memory contains $\sigma = a_0 a_1 \cdots a_{j-1} a_j$. At the next cycle the ACM will contain a sequence of data σ' such that one of the following is true:

1. $\sigma' = \sigma$: neither the reader has removed or the writer has stored any data item in σ;

2. $\sigma' = a_0 a_1 \cdots a_{j-1} a_j a_{j+1}$: the reader has not removed any item from σ, but the writer has added a new item;

3. $\sigma' = a_1 \cdots a_{j-1} a_j$: the reader or the writer has removed a data item from the head of σ.

4. $\sigma' = a_0 a_2 \cdots a_{j-1} a_j$: the writer has removed a data item from the head of σ.

The above can be specified by the following CTL formula:

$$\begin{aligned} \mathbf{AG}(|\sigma| = x \rightarrow \mathbf{AX}((|\sigma'| >= x \wedge \sigma' = \sigma^+) \vee \\ (|\sigma'| = x - 1 \wedge \sigma' = \sigma^-))) \end{aligned}$$

where σ^+ is used to denote $a_0 \cdots a_j$ or $a_0 \cdots a_j a_{j+1}$ and σ^- is used to denote $a_1 \cdots a_{j-1} a_j$ or $a_0 a_2 \cdots a_{j-1} a_j$. Observe that 1 and 2 are captured by the same same CTL sub-formula, which is given by the left side of the $\vee$ inside the **AX** operator.

3 Modeling ACMs with CPN

In this section we will describe an ACM fitting the behavior set introduced in Section 2 with a Hierarchical Colored Petri Net (HCPN) [Jen92, Jen97] model. We will then describe the properties of data coherence and data freshness using ASKCTL and verify the HCPN model of the ACM for these properties.

An HCPN is a set of non-hierarchical CPN models in which each model is called a *page*. Two mechanisms are introduced to allow hierarchical levels: substitution transition and fusion places. A substitution transition is a transition that represents a CPN page. The fusion places are physically different but logically the are the same, defined by means of a fusion set. All places belonging to a fusion set have the same marking. As in other types of Petri nets models, a marking of a place is the set of tokens in that place at a given moment. And the marking of a net is the set of markings of all places. When a marking of a place belonging to a fusion set changes, the marking of all places belonging to that set also changes.

In order to manipulate tokens in a CPN, it is defined the concept of multi-set. A multi-set is a set where it is possible to have several occurrences of the same element. This concept allows similar parts of the model to be modeled as token information instead of structure replication.

Figures 4 and 5 shows the HCPN models for the writer and reader processes as introduced by Definition 1, respectively. Each process has two transitions, one modeling the beginning of a buffer access

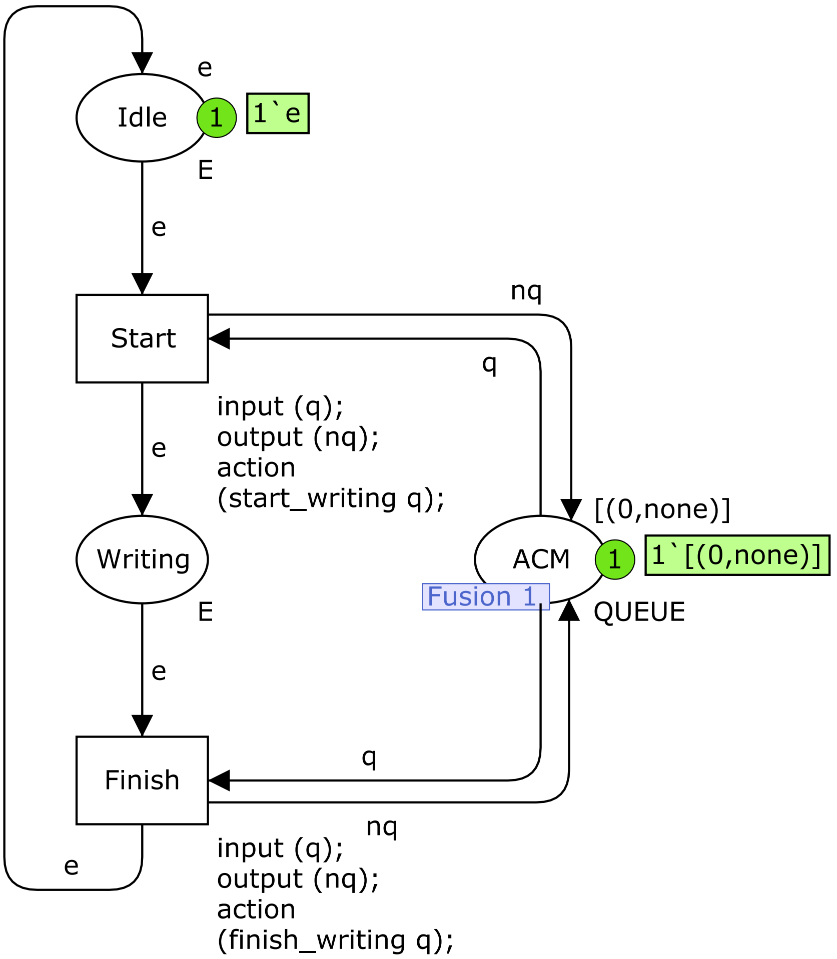

Figure 4: CPN model for the writer

action and other modeling the end of the action. In the initial state both processes are ready to initiate an access action and the buffer is initialized with some data.

Places labeled *ACM* in the page of the writer and in the page of the reader model the queue of data σ. These places belong to the same fusion set, meaning that they are the same place. For this reason we will not distinguish them from now. The type of the tokens in *ACM* is a list of data items. A data item is a pair **(data,status)** where **data** is the data being transmitted and **status** is one of **wr**, **rd** or **none**. Indicating if the data is being written or read in a given marking. To avoid the state space explosion, we used **data** as a Boolean, but it can be set as an integer, string or any other data type.

When the writer begins writing some data into the ACM, modeled by transition *Start* in Figure 4, it adds a pair **(data_value, wr)** to the end of the token already stored in place *ACM*. For instance, if the current marking of *ACM* is [**(false, none)**] and the value to be transmitted is **true**, then after the occurrence of *Start* the marking of *ACM* will be [**(false,none),(true,wr)**]. In the notation of the previous Section we have that $\langle false\rangle \xrightarrow{wr_b(true)} \langle false\ true^{wr}\rangle$.

In a similar way, when the writer finishes accessing the ACM, modeled by transition *Finish*, it updates the value of the token in place *ACM* to indicate that the new value is available for reading. In the example above, the new marking of *ACM* will be [**(false,none),(true,none)**].

A similar reasoning applies to the reader process. When it starts reading from the ACM the value of the first element of the token on place *ACM* is modified to indicate the beginning of the access. More specifically, if the value of the token is [**(false,none),(true,none)**], after the occurrence of transition *Start* of the reader process, the new marking of *ACM* will be [**(false,rd),(true,none)**]. If the buffer is full, some data should be replaced to proceed writing a new one. This is done by the function `start_writing()` called in the code region of transition *Start*.

When the reader finishes the data read action, depending on the status of the queue the first element of the list of data items is removed. Again, if the list of data items contains [**(false,rd),(true,none)**], the head of the list can be removed and the new data queue will contain [**(true,none)**]. However, if the ACM contains [**(false,rd)**] or [**(false,rd), (true,wr)**], then the head cannot be removed without the risk of both processes addressing the same memory segment. In this case the reader will prepare to re-read the head of the queue, and the marking of place *ACM* will not change. In any case, the reader is not required to wait for some event from the writer. Observe that neither process is required to wait for the other in any situation

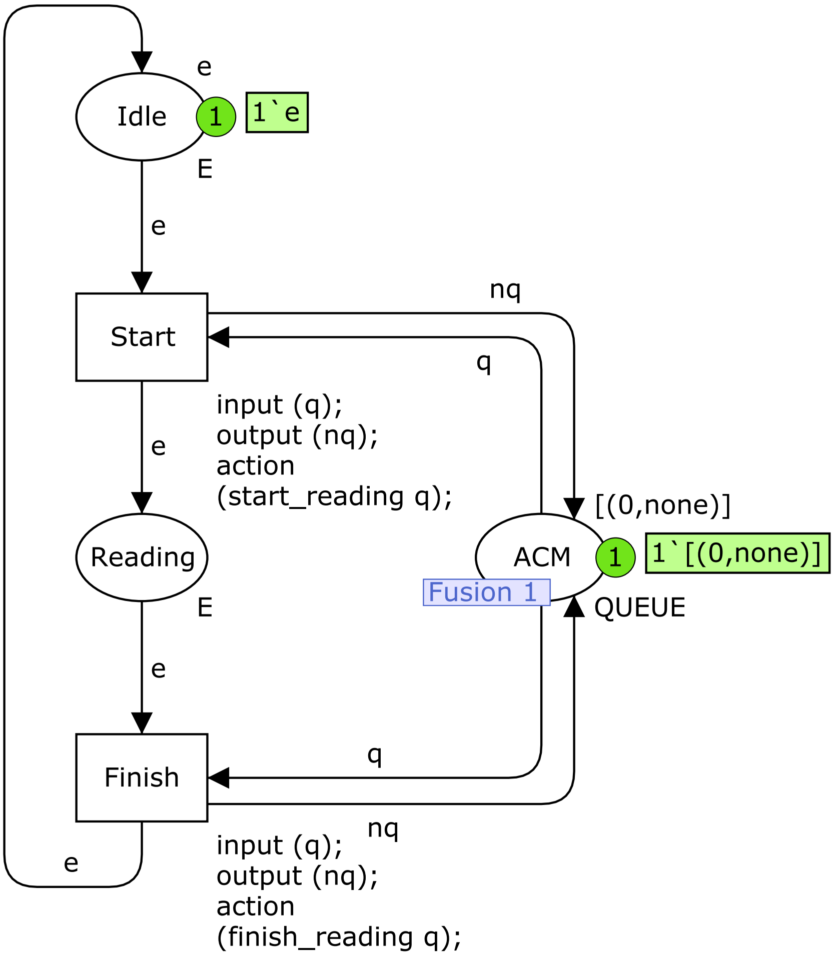

Figure 5: CPN model for the reader

Observe that for both processes the modifications of the value of a data item are executed by the SML functions associated to each transition of the processes. These functions are:

1. `start_writing(q:QUEUE)`
2. `finish_writing(q:QUEUE)`
3. `start_reading(q:QUEUE)`
4. `finish_reading(q:QUEUE)`

The function `start_writing(q:QUEUE)` is responsible for adding an item of the type **(DATA, wr)** to the token representing the status of the buffer. The source code bellow shows such a function.

```
fun start_writing data: QUEUE =
2   (
3   let
4     val wdata = discrete(min, max);
5     val mesg = "start writing "^Int.toString(wdata);
6   in
7     msc.addEvent("writer", "reader", mesg);
8     if (length data = n andalso
9         (#2 (hd data) = rd)) then (
10      [hd data]^^(tl (tl data))^^[(wdata, wr)]
11    ) else if (length data = n andalso
12                (#2 (hd data) = none)) then
13      tl(data)^^[(wdata, wr)]
14    else
15      data^^[(wdata, wr)]
16  end
17  );
```

Two constants are declared: one representing the new data to be transmitted, given by `wdata`; other to define the message that appears on the MSC. The function first generates the message that appears on the MSC (line 7), then it checks if the size of the available data queue (i.e. the data that has not been read) is equal to the size of the ACM or not (line 8). If this is true and the reader is accessing the ACM (line 9) then the second data item on the queue is removed and the writer starts putting a new data on the end of the queue (line 10). This code implements the behavior defined by rule 3 on Definition 1.

On the other hand, if the queue is full but the reader is not accessing the ACM (lines 11 and 12), then the first item is discarded and writer starts storing a new data on the end of the queue (line 13). And this implements the behavior of rule 2 of Definition 1. Finally, if the queue is not full, the writer simply starts storing a new data in the end of the queue (lines 14 and 15), which implements rule 1 of Definition 1.

The function `finish_writing(q:QUEUE)` is responsible for changing the last item in the buffer from **(DATA, wr)** to **(DATA, none)** indicating that the new data was released for reading. Its source code is shown bellow.

```
fun finish_writing data:QUEUE =
(
let
  val mesg = "finish writing";
in
  if (length (tl data) > 0) then
    [hd data]^^finish_writing(tl data)
  else (
    msc.addEvent("writer", "reader", mesg);
    [(#1 (hd data), none)]
  )
end
);
```

The writer is always allowed to release the new item. `finish_writing` simply signals on the last data item that it has finished writing by replacing the pair `(data, wr)` by `(data, none)`. Note that this is a recursive function, then it needs to check for the size of the queue until it is empty (i.e. the data queue has only its head). `finish_writing` implements the behavior defined by rule 4 of Definition 1.

The function `start_reading(q:QUEUE)` is responsible for changing the first item in the buffer from **(DATA, none)** to **(DATA, rd)** to indicate that the reader started accessing it.

```
fun start_reading data: QUEUE =
(
let
  val mesg = "start reading "^Int.toString(#1 (hd data));
in
  msc.addEvent("reader", "writer", mesg);
  [(#1 (hd data), rd)]^^tl data
end
);
```

The function `finish_reading(q:QUEUE)` is responsible for determining if the reader will next re-read the current data item or get a new one. If re-read is triggered, it changes the first item in the buffer from **(DATA, rd)** to **(DATA, none)**, otherwise it removes that item from the buffer.

```
fun finish_reading data: QUEUE =
(
let
  val mesg = "finish reading";
in
  msc.addEvent("reader", "writer", mesg);
  if (length data = 1) then
    [(#1 (hd data), none)]
  else if (#2 (hd (tl data)) = wr) then
    [(#1 (hd data), none)]^^tl data
  else
    tl data
end
);
```

The behavior of `start_reading` and `finish_reading` can be inferred from the description of the behavior of the writer related functions. `start_reading` implements the behavior defined by rule 5 while `finish_reading` implements the behavior of rule 6 to 8 of Definition 1.

Note that the writer can also remove items from the queue when overwriting it. This is done by the function `start_writing()` when the buffer is full of non read data and the oldest non read data item is overwritten. In such cases, the second or the first item in the buffer is replaced according to either the reader is accessing the buffer or not. It is also interesting to notice that in the writing functions the focus is on the act of start writing, while in the reading functions the focus is on finishing reading. This is due to the fact that the writer has no restrictions about releasing a new item, while the reader has no restriction about getting the item it is prepared to get.

4 Validation and verification

In order to illustrate the behavior of the CPN model introduced above a number of Message Sequence Charts (MSC) [HT03] has been automatically generated from the simulation of the model. For instance, in Figure 6 the MSC generated when the reader and writer processes are about the same speed is showed. In this case re-reading and overwriting do not occur.

In such MSC the message labeled **start writing 100** is generated by the transition *Start* of the writer process and it indicates that it is starting to write the value **100** in the buffer. On the same way, the

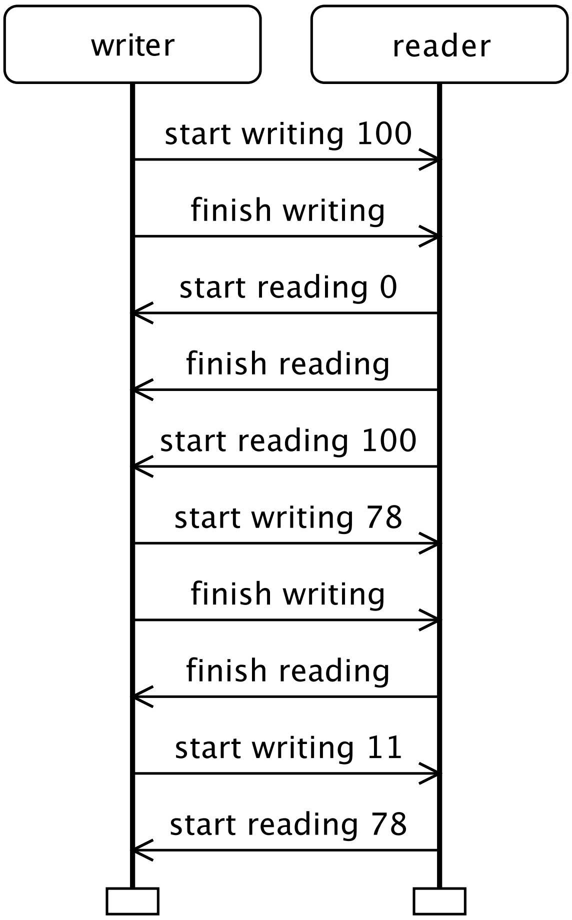

Figure 6: MSC for no re-reading and no overwriting case

message **finish writing** indicates that the writer is releasing the new data for reading operation. The same reasoning applies for messages generated by the reader.

An important observation that should be done is that the labels of the messages in the chart do not reflect the information exchanged by the processes. Such labels are abstractions of the changes in the token containing the data being communicated, i.e. the token in place *ACM*. In the real implementation these messages are replaced by changes in values of the control variables.

In the initial state the buffer is initialized with the data value **0** and none of the processes are accessing it. According to the MSC the following sequence of states, as introduced in Definition 1, is generated:

$$\begin{aligned}
\sigma_0 &= 0\\
\sigma_1 &= 0\ 100^w\\
\sigma_2 &= 0\ 100\\
\sigma_3 &= 0^r\ 100\\
\sigma_4 &= 100\\
\sigma_5 &= 100^r\\
\sigma_6 &= 100^r\ 78^w\\
\sigma_7 &= 100^r\ 78\\
\sigma_8 &= 78\\
\sigma_9 &= 78\ 11^w\\
\sigma_{10} &= 78^r\ 11^w
\end{aligned}$$

Observe that the sequence of data received by the reader (0, 100, 78...) is the same that was sent by the writer (0, 100, 78, 11...), except for the last data that has not been received yet. Also, note that any σ_i can be easily mapped into a token contained in the place *ACM*. For instance, 0 100^w is mapped into the token [**(0, none), (100, wr)**]

In Figure 7 the MSC generated when re-reading occurs is illustrated. In this case reader access the buffer twice, recovering the data value **0** on both, before the writer access it for the first time. Observe that the writer accesses the buffer before the reader finishes its second operation. For this reason, on the next time the reader recovers the new value **100**, otherwise it should engage in another re-reading operation.

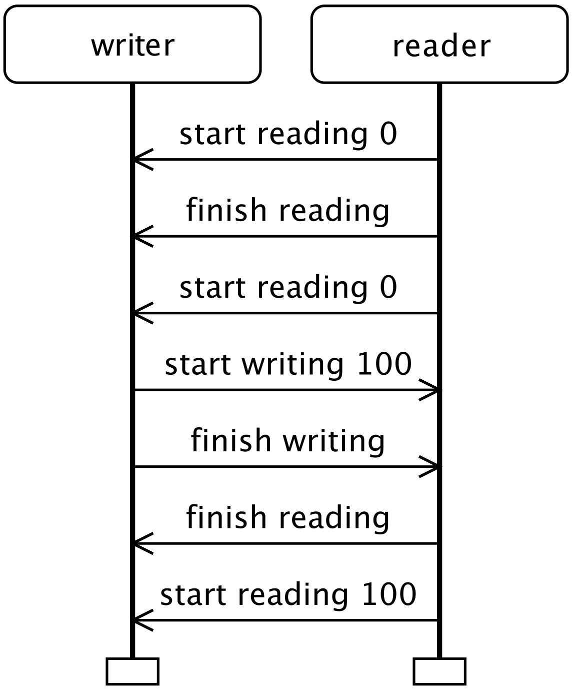

Figure 7: MSC for the re-reading case

For this sequence of messages, the sequence of states according to Definition 1 is as follows:

$$
\begin{aligned}
\sigma_0 &= 0\\
\sigma_1 &= 0^r\\
\sigma_2 &= 0\\
\sigma_3 &= 0^r\\
\sigma_4 &= 0^r\ 100^w\\
\sigma_5 &= 0^r\ 100\\
\sigma_6 &= 100\\
\sigma_7 &= 100^r
\end{aligned}
$$

Finally, in Figure 8 the MSCs obtained when overwriting occurs is introduced. In this case the writer attempts to send another data item when the buffer is already full of items. The writer first sends the value **100** and just after that it sends the value **78**. On its second operation, the value **100** is replaced due to the fact that the reader is already accessing the memory position containing **0**. This is needed in order to preserve data coherence. Note that in this case the ACM can hold at most two at a time. In this case the sequence of states is as follows:

$$
\begin{aligned}
\sigma_0 &= 0\\
\sigma_1 &= 0^r\\
\sigma_2 &= 0^r\ 100^w\\
\sigma_3 &= 0^r\ 100\\
\sigma_4 &= 0^r\ 78^w\\
\sigma_5 &= 0^r\ 78\\
\sigma_6 &= 78\\
\sigma_7 &= 78^r
\end{aligned}
$$

As can be observed the MSCs generated by the model discussed above reflect the behavior set defined by the transition system introduced by Definition 1. The MSCs are not a proof of correctness of the model. However they give a good intuition that the model is correct with respect to Definition 1. Besides that, they promote an intuitive way of understanding how the ACM policies work.

Nevertheless, in order to give formal arguments of the correctness of the model, coherence and freshness properties introduced in Section 2 were modeled using the ASKCTL model checker. ASKCTL is a model checker originally designed to run inside the Design/CPN tools that is also embedded into CPN-Tools [JKW07, RWL⁺03]. In ASKCTL, model checking requires the generation of the occurrence graph of the CPN model and then the generation of its strongly connected components graph. Temporal logic formulae are described as a CTL like language and the atomic propositions are described by SML functions. Each of these functions should receive as input a node of the occurrence graph and evaluate it to true or false. Describing the formulae is a question of using the correct syntax.

In ASKCTL, the operator → is not available, so it is necessary to rewrite the part of the CTL formula for coherence (see Section 2) that is in the form $A \rightarrow B$, to its equivalent $\neg A \vee B$. Also, it is necessary to use the ASKCTL operator equivalent to `AG`. Finally, the atomic propositions are written as SML functions receiving a node from the state space as parameter and returning a Boolean. The following ASKCTL formula is then obtained:

```
INV(OR(NOT(NF("reading", has_rd)),
       AND(NF("reading head", rd_first),
```

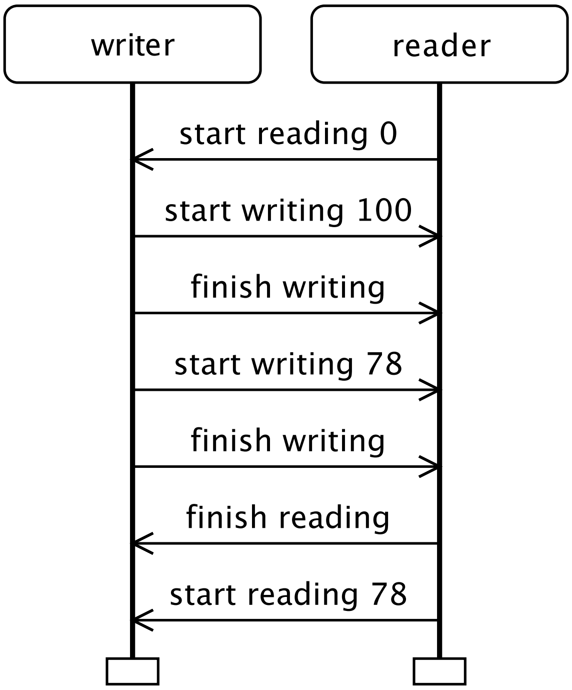

Figure 8: MSC for the overwriting case

```
              OR(NOT(NF("writing", has_wr)),
                 NF("writing last", wr_last)))));
```

In the above, INV is the ASKCTL equivalent of AG and has_rd, rd_first, has_wr and wr_last are the SML functions for the atomic propositions. has_rd and has_wr check if the reader and the writer are accessing the ACM at a given state, respectively. While rd_first and wr_last checks if the reader is accessing the first position of the queue and if the writer is accessing the last, in the case that some of them is accessing it. NF is used to tell ASKCTL that the proposition reefers to node of the state and not to an arc. The formula above is enough to verify ACMs of any size.

Freshness is much more complicated. And it requires one formula for each possible size the data queue may have, i.e. from 1 to n assuming that n is the size of the ACM. For instance, the formula describing freshness for the queue holding 2 items is given by:

```
INV(OR(NOT(NF("lenght of sigma is 2", check_length 2)),
       FORALL_NEXT(OR(AND(OR(NF("|sigma'| = 2", check_length 2),
                             NF("|sigma'| = 2+1", check_length 3)),
                          NF("sigma' = sigma+", check_sigma_plus)),
                      AND(NF("|sigma'| = 2-1", check_length 1),
                          NF("sigma' = sigma-", check_sigma_less))))));
```

Again, it is necessary to re-write the $A \rightarrow B$ and replace the operator AG and AX by its ASKCTL equivalents. Which are INV and FORALL_NEXT respectively. check_length checks if the size of the data queue equals to some integer, check_sigma_plus checks if the new queue is on the form defined by σ^+ and check_sigma_less if it is of the form σ^- as defined in Section 2.

One important observation that should be made here is that it is not possible to verify coherence without the information about the previous state of the data queue. So, to check for coherence a small modification is made to the model. A new place is added to the CPN to store the previous value of the data queue. Every time a transitions fires, it backs up the token on place *ACM*. On Figure 9 this modification is showed.

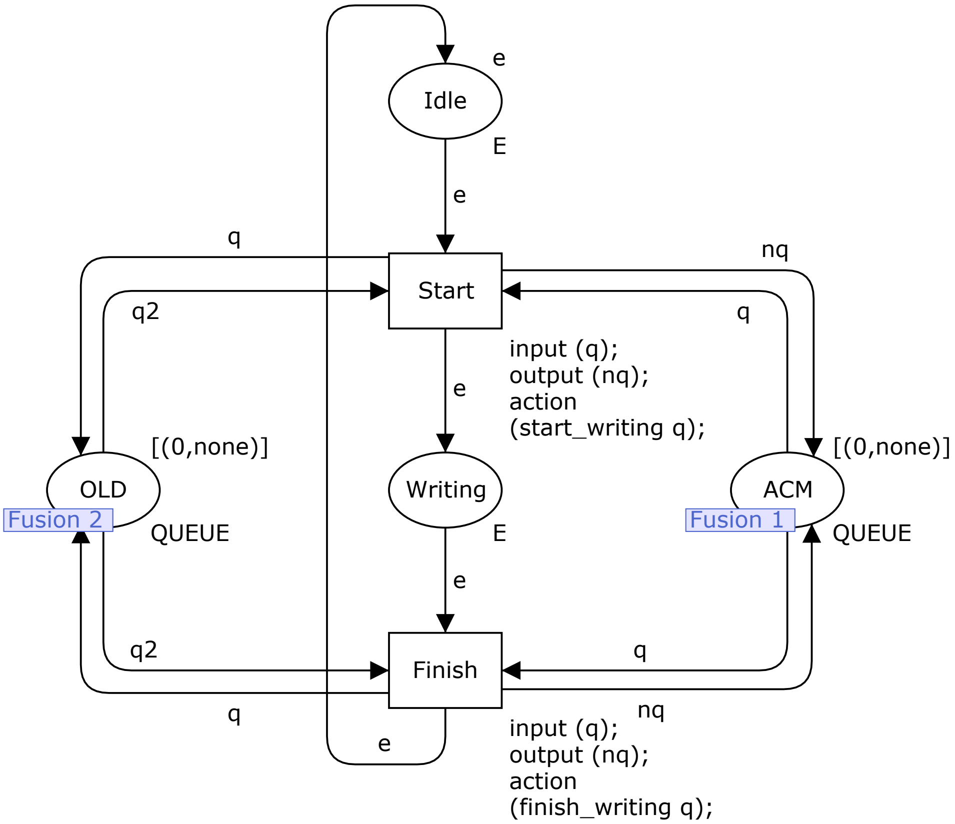

Figure 9: CPN model for the writer saving queue state

Both properties were verified and proved to be correct for a number of OWRRBB ACMs of different sizes. Observe that the model introduced here is not generic in the sense of a model for OWRRBB ACMs of any size. However, it is parametrized and modifying the size of the ACM is a very simple operation.

Besides coherence and freshness we have also verified that the initial marking of the CPN model is a home marking, meaning that the system can always return to its initial state if the set of allowed data values is finite.

5 Conclusions and future work

Related work has introduced automatic methods for the synthesis of ACMs. The synthesis process starts from a functional specification and concludes with an implementation of the ACM. At the moment it is possible to generate C++ source code to communicate two processes asynchronously. In this work, one class of ACMs is formally defined and modeled as an HCPN. Then it is verified against a set of properties described in temporal logic. Although OWRRBB ACMs have been designed, synthesized and verified before, they all concentrated in single-cell solutions. This is the first attempt at designing and verifying a multi-cell OWRRBB.

Firstly, the behavior of OWRRBB ACMs was formally defined and modeled as an HCPN using the CPNTools. The properties that such ACM should satisfy were also discussed and described by CTL formulae. The specified properties include: coherence, freshness, reversibility and absence of deadlocks. All properties were proved to be satisfied. However, when the size of the ACM grows, it is more difficult to perform model checking due to the state explosion problem. Our attempts at verifying at a high level only delays this problem. Additionally, the OWRRBB CPN model was used to generated a number of MSCs that were used to validate the model with respect to the behavior set introduced by Definition 1. This validation was performed manually. However, the methodology introduced can be used to increase the confidence of the designer when specifying a new protocol, and it is done at a higher level of abstraction, without the need of generating a low-level algorithmic description. Finally, the MSCs are generated automatically through the simulation of the CPN model.

Plans for future work include providing a formal proof that all ACMs with a behavior compatible with the formal behavior we have defined satisfies coherence and freshness. This step is needed due to the fact that our models are not models of ACMs of any size. Model checking is performed for some size n and nothing can be assured for size $n+1$ unless model checking is performed again. With such proof, no model checking is needed at all. Also, we intend to use the CPN models to generate an implementation that cannot be trusted unless we have formally verified the model. Since the state explosion problem cannot be avoided, only minimized, a proof is of great importance.

At the present time the generation of C++ code from low-level Petri nets has been introduced. Another future work is to extend that method to generate code from high-level Petri net such as CPNs. Finally, it is a primary goal to be able to generate the ACMs in the form of a Verilog code that can be used to synthesize a piece of hardware.

Acknowledgments

This work is supported by the EPSRC through projects NEGUS (EP/C512812/1) and STEP (EP/E044662/1) at Newcastle University, by Nokia Institute of Technology/Nokia do Brasil through cooperation project with Federal University of Campina Grande and by CICYT TIN2004-07925, a Distinction for Research from the Generalitat de Catalunya.

References

[Fas01] Jean-Philippe Fassino. *THINK: vers une architecture de systèmes flexibles*. PhD thesis, École Nationale Supérieure des Télécommunications, December 2001.

[GCX07] Kyller Gorgônio, Jordi Cortadella, and Fei Xia. A compositional method for the synthesis of asynchronous communication mechanisms. In Jetty Kleijn an Alex Yakovlev, editor, *ICATPN*, number 4546 in LNCS, pages 144–163. Springer-Verlag Berlin Heidelberg, 2007.

[GCXY07] Kyller Gorgônio, Jordi Cortadella, Fei Xia, and Alex Yakovlev. Automating synthesis of asynchronous communication mechanisms. *Fundamenta Informaticae*, 78(1):75–100, June 2007.

[HT03] David Harel and P. S. Thiagarajan. Message sequence charts. pages 77–105, 2003.

[Jen92] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 1 of *EACTS – Monographs on Theoretical Computer Science*. Springer-Verlag, 1992.

[Jen97] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 2 of *EACTS – Monographs on Theoretical Computer Science*. Springer-Verlag, 1997.

[JKW07] Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *Int. J. Softw. Tools Technol. Transf.*, 9(3):213–254, 2007.

[Lam86] Leslie Lamport. On interprocess communication — parts I and II. *Distributed Computing*, 1(2):77–101, 1986.

[Mur89] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[RWL$^+$03] Anne V. Ratzer, Lisa Wells, Henry Michael Lassen, Mads Laursen, Jacob Frank Qvortrup, Martin Stig Stissing, Michael Westergaard, Søren Christensen, and Kurt Jensen. Cpn tools for editing, simulating, and analysing coloured petri nets. In *ICATPN*, pages 450–462, 2003.

[Sim03] Hugo R. Simpson. Protocols for process interaction. *IEE Proceedings on Computers and Digital Techniques*, 150(3):157–182, May 2003.

[XHC$^+$04] Fei Xia, Fei Hao, Ian Clark, Alex Yakovlev, and Graeme Chester. Buffered asynchronous communication mechanisms. In *Proceedings of the Fourth International Conference on Application of Concurrency to System Design (ACSD'04)*, pages 36–44. IEEE Computer Society, 2004.

[YKXK98] Alex Yakovlev, David J. Kinniment, Fei Xia, and Albert M. Koelmans. A fifo buffer with non-blocking interface. *TCVLSI Technical Bulletin*, pages 11–14, Fall 1998.