A compositional method for the synthesis of Asynchronous Communication Mechanisms

Kyller Gorgonio[1], Jordi Cortadella[2], and Fei Xia[3]

[1] Embedded Systems and Pervasive Computing Laboratory
Federal University of Campina Grande, Brazil
[2] Department of Software
Universitat Politècnica de Catalunya, Spain
[3] School of Electrical, Electronic and Computer Engineering
University of Newcastle upon Tyne, UK

Abstract. Asynchronous data communication mechanisms (ACMs) have been extensively studied as data connectors between independently timed concurrent processes. In previous work, an automatic ACM synthesis method based on the generation of the reachability graph and the theory of regions was proposed. In this paper, we propose a new synthesis method based on the composition of Petri net modules, avoiding the exploration of the reachability graph. The behavior of ACMs is formally defined and correctness properties are specified in CTL. Model checking is used to verify the correctness of the Petri net models. The algorithms to generate the Petri net models are presented. Finally, a method to automatically generate C++ source code from the Petri net model is described.
keywords: Asynchronous communication mechanisms, Petri nets, concurrent systems, synthesis, model checking, protocols.

1 Introduction

One of the most important issues when designing communication schemes between asynchronous processes is to ensure that such schemes allow as much asynchrony as possible after satisfying design requirements on data. When the size of computation networks becomes large, and the traffic between the processing elements increases, this task becomes more difficult.

An *Asynchronous Communication Mechanism* (ACM) is a scheme which manages the transfer of data between two processes, a *producer* (writer) and a *consumer* (reader), not necessarily synchronized for the purpose of data transfer. The general scheme of an ACM is shown in Figure 1. It includes a shared memory to hold the transferred data and control variables. In this work it is assumed that the data being transferred consists of a stream of items of the same type, and the writer and reader processes are single-threaded loops. At each iteration a single data item is transferred to or from the ACM.

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 [4].

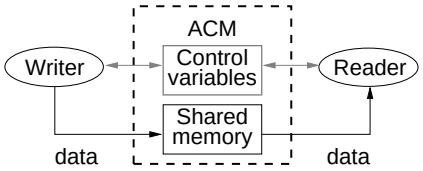


Fig. 1. ACM with shared memory and control variables.

By using single-bit unidirectional variables, the synchronization control can be reduced to the reading and writing of these variables by extremely simple atomic actions [6]. Variables are said to be *unidirectional* when they can only be modified by one of the processes. This provides the safest solution for a maximum asynchrony between the writer and the reader. In particular, if the setting, resetting and referencing of control variables can be regarded as atomic events, the correctness of ACMs becomes easy to prove.

ACMs are classified according to their *overwriting* and *re-reading* policies [8,6]. Overwriting occurs when the ACM is full of data that has not been read before. In this case the producer can overwrite some of the existing data items in the buffer. Re-reading occurs when all data in the ACM has been read by the consumer. 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 is 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.

The choice of using a particular class of ACM for a particular job is generally based on data requirements and system timing restrictions [4,6]. 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 in the buffer [6,9,2]. Overwriting the newest item in the buffer [9] attempts to provide the reader with the best continuity of data items for its next read. 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.

1.1 ACM example

Now consider an RRBB 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 to 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 2.

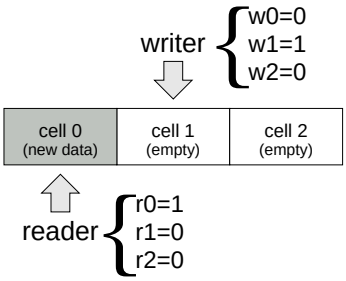


Fig. 2. Execution of RRBB ACM with 3 cells.

The writer always stores some data into the ACM and then attempts to advance to the next cell releasing the new data. In this way, 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". A similar behavior applies to the reader. A possible trace for the reader is $\langle rd_0 rd_1 rd_1 rd_2 \rangle$.

In an RRBB ACM, no overwriting and allowing re-reading imply the following behavior:

- The writer first accesses the shared memory and then advances to the next cell, but only if the reader is not pointing at it.
- The reader first advances to the next cell if the writer is not there and then performs the data transfer, otherwise it re-reads the current cell.

In general, and depending on how the read and write 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 trace for this system 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 introduces the problem of data coherence when the reader and the writer are retrieving and storing data on the same memory locations.

Freshness is related to the fact that the last data record produced by the writer must be available for the reader. On the ACMs studied in this work, the reader always attempts to retrieve the oldest data stored in the shared memory that has not been read before. This means that the freshness property imposes a specific sequencing of data, i.e. the data is read in the same order that it is written. Depending on the ACM class, some data may be read more than once or may be missed. However, the sequence should be preserved. For the example above, one possible trace is $\langle wr_1 rd_0 wr_2 rd_1 rd_1 \cdots \rangle$. Note that 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, which violates freshness.

With a correct interleaving both processes will avoid accessing the same data cell at the same time, the writer will not be allowed to overwrite unread data, and the reader will have the possibility of re-reading the most recent data only when there is no unread data in the ACM. For the example above, a correct trace is $\langle wr_1 rd_0 rd_1 wr_2 rd_1 wr_0 rd_2 wr_1 \rangle$. Observe that 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 on a re-reading regardless the actions of the writer.

Algorithm 1 RRBB ACM with 3 cells

```
Require: Boolean w_0, w_1, w_2
Require: External Boolean r_0, r_1, r_2
 1: process writer()
 2:    w_1 := 1; w_0 := w_2 := 0;
 3:    loop
 4:       if w_0 = 1 ∧ r_1 = 0 then
 5:          write cell 1;
 6:          w_0 := 0; w_1 := 1;
 7:       else if w_1 = 1 ∧ r_2 = 0 then
 8:          write cell 2;
 9:          w_1 := 0; w_2 := 1;
10:       else if w_2 = 1 ∧ r_0 = 0 then
11:          write cell 0;
12:          w_2 := 0; w_0 := 1;
13:       else
14:          wait until some r_i is modified;
15:       end if
16:    end loop
17: end process
```

```
Require: Boolean r_0, r_1, r_2
Require: External Boolean w_0, w_1, w_2
 1: process reader()
 2:    r_0 := 1; r_1 := r_2 := 0;
 3:    loop
 4:       if r_0 = 1 ∧ w_1 = 0 then
 5:          r_0 := 0; r_1 := 1;
 6:          read cell 1;
 7:       else if r_0 = 1 ∧ w_1 = 1 then
 8:          read cell 0;
 9:       else if r_1 = 1 ∧ w_2 = 0 then
10:          r_1 := 0; r_2 := 1;
11:          read cell 2;
12:       else if r_0 = 1 ∧ w_1 = 1 then
13:          read cell 1;
14:       else if r_2 = 1 ∧ w_0 = 0 then
15:          r_2 := 0; r_0 := 1;
16:          read cell 0;
17:       else if r_2 = 1 ∧ w_0 = 1 then
18:          read cell 2;
19:       end if
20:    end loop
21: end process
```

A possible implementation of the example above is described in Algorithm 1. The writer is shown on the left side and the reader on the right. Each process consists of an infinite loop. This is just a simple abstraction of the real behavior of a process, in which the ACM operations are combined with the data processing actions. 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. Note that when the writer is waiting, 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 (with or without successfully advancing) the reader then reads (or rereads) from cell r. Note that 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.

In the rest of this paper, a Petri net based method for the automatic synthesis of ACMs is presented. The method receives as input a functional specification consisting of the ACM class that should be implemented by the ACM and the number of cells it should have. As output, it produces the source code implementing the operations with the ACM. The implementation can be either in software (e.g. C++ or Java) or hardware (e.g. Verilog or VHDL).

In this paper, we will provide C++ implementations. For instance, the C++ code for the 3-cell RRBB ACM described above is shown in Figures 6 and 7.

In the next sections, the methodology presented in the paper will be described. The behavior of the RRBB ACM class will be formally defined and the method to generate its implementation will be detailed. Due to the limited space, the OWBB and OWRRBB classes will not be discussed in detail. However, the principle used to generate the RRBB implementations also applies to the overwriting ACM classes.

2 Overview of the approach

In previous work [1,8,10], a step-by-step method based on the theory of regions for the synthesis of ACMs was presented. The method required 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. Next, a Petri net model was obtained using the concept of ACM regions, a refined version of the conventional regions.

This work proposes the generation of the 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 abutting a set of Petri net modules. The correctness of the model can then be formally verified using model checking. The relevant properties of the ACM, coherence and freshness, can be specified using CTL formulae. This paper also extends previous work by introducing an approach to automatically generate the implementation of the ACM from the Petri net model. Figure 3 shows the design flow for the automatic generation of ACMs.

Compared to the previous work, the new approach has the advantage of not dealing with the entire state space of the ACM when generating the Petri net

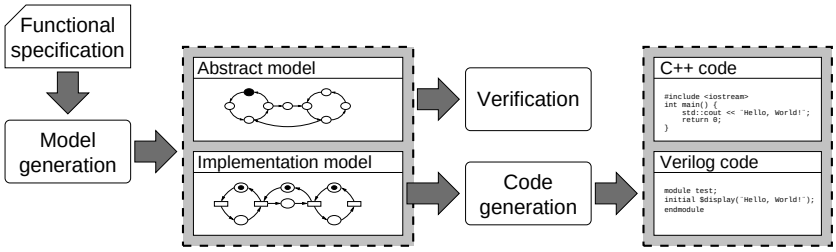


Fig. 3. The design flow

model. It is obtained in linear time. On the other hand, it requires to verify the model generated to provide enough evidence of its correctness. Observe that it is possible to obtain the ACM implementation without doing verification. In practice, the new approach allows to obtain the Petri net model when the size of the ACM grows.

2.1 Models for verification and implementation

The two basic paradigms on the approach presented in this paper are *automation* and *correctness*. For that reason, from the functional specification of an ACM, two formal models are generated:

- An *abstract model*, that describes the possible traces of the system and that is suitable for model checking of the main properties of the ACM: coherence and freshness. These properties can be modeled using temporal logic formulae.
- An *implementation model*, that is suitable for generating a hardware or software implementation of the ACM. This model is generated by the composition of basic Petri net modules and contains more details about the system. This model is required to narrow the distance between the behavior and the implementation.

For a complete verification of the system, a bridge is required to check that the implementation model is a refinement of the abstract model. For such purpose, the Cadence SMV Model Checker [5] has been used.

The Cadence SMV extends the CMU SMV model checker by providing a more expressive description language and by supporting a variety of techniques for compositional verification. In particular, it supports refinement verification by allowing the designer to specify many abstract definitions for the same signal. It can then check if the signal in a more abstract level is correctly implemented by another abstraction of lower level.

Thus, the correctness of the generated ACMs is verified as follows:

1. The abstract and implementation models of the ACM are generated.
2. The properties of the ACM are specified in CTL and model checked on the abstract model.

3. The implementation model is verified to be a refinement of the abstract model.

In the forthcoming section, the abstract and implementation models for the class of RRBB ACMs are presented.

3 The abstract model for RRBB ACMs

The abstract model for an RRBB ACM is specified as a transition system. The state of the ACM is defined by the data items available for reading. For each state, σ defines the queue of data stored in the ACM. More specifically, σ is a sequence: $\sigma = a_0 a_1 \cdots a_{j-1} a_j$, with $j < n$, where n is size of the ACM. 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 must also express if the processes are accessing the ACM or not. This is done by adding flags to 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.

Observe that σ can be interpreted as a stream of data that is passed from the writer (on the left) to the reader (on the right). 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 $\langle \sigma_j \rangle$, whereas $\langle \sigma \rangle \xrightarrow{e} \perp$ is used to denote that e is not enabled in $\langle \sigma \rangle$.

In RRBB ACMs, the reader is required not to wait when starting an access to the ACM. In the case there is no new data in the ACM, the reader will re-read some data that was read before.

The writer can add data in the ACM until it is full. In such case, the writer is required to wait until the reader retrieves some data from the ACM. 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.

Definition 1 formally captures the behavior of RRBB ACMs. Rules 1-3 model the behavior of the writer. Rules 4-7 model the behavior of the reader.

Definition 1 (RRBB transition rules) *The behavior of an RRBB 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. $\langle \sigma \rangle \xrightarrow{wr_b(a)} \langle \sigma a^w \rangle$ **if** $|\sigma| < n$
2. $\langle \sigma \rangle \xrightarrow{wr_b(a)} \perp$ **if** $|\sigma| = n$
3. $\langle \sigma a^w \rangle \xrightarrow{wr_e(a)} \langle \sigma a \rangle$
4. $\langle a\sigma \rangle \xrightarrow{rd_b(a)} \langle a^r \sigma \rangle$
5. $\langle a^r \sigma \rangle \xrightarrow{rd_e(a)} \langle \sigma \rangle$ **if** $|\sigma| > 0 \wedge \sigma \neq b^w$
6. $\langle a^r \rangle \xrightarrow{rd_e(a)} \langle a \rangle$
7. $\langle a^r b^w \rangle \xrightarrow{rd_e(a)} \langle a b^w \rangle$

Rule 1 models the start of a write action for a new data item a and signaling that it is not available for reading (a^w). Rule 3 models the completion of the write action and making the new data available for reading. Finally, rule 2 represents the blocking of the writer when the ACM is full ($|\sigma| = n$).

Rule 4 models the beginning of a read action retrieving data item a and indicating that it is being read (a^r). Rule 5 models the completion of the read operation. In this rule, a is removed from the buffer when other data is available. On the other hand, rules 6 and 7 model the completion of the read action when no more data is available for reading. In this case, the data is not removed from the buffer and is available for re-reading. This is necessary due to the fact that the reader is required not to be blocked even if there is no new data in the ACM.

It is important to observe that in the state $\langle a^r b^w \rangle$ the next element to be retrieved by the reader will depend 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.

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 4 to verify a lower level specification of the ACM. Next, the specification of the coherence and freshness properties is discussed.

3.1 Coherence

To verify the coherence property it is necessary to prove that there is no reachable state in the system in which both processes are addressing the same segment of the shared memory.

In the ACM model described by Definition 1, the reader always addresses the data stored in the first position of the ACM, represented by σ. On the other hand, the writer always addresses the tail of the ACM. To prove coherence in this model it is only necessary to prove that every time the reader is accessing the ACM, then:

- it is addressing the first data item, and
- 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:

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

The formula above specifies that for any reachable state of the system (AG), if the reader is accessing the ACM, then:

1. It is reading a data from the beginning of the buffer ($a^r = a_0$);
2. If the writer is also accessing the ACM, then it is not pointing at the beginning of the queue ($(a^w \in \sigma \rightarrow a^w = a_j \wedge j > 0)$).

3.2 Freshness

As discussed before, freshness is related to sequencing of data. Now, let us assume that at a certain time the shared memory contains a sequence of data $\sigma = a_0 a_1 \cdots a_{j-1} a_j$, with $j < n$, a_j is the last written data, and a_0 is the next data to be retrieved by the reader. Then, at the next cycle the ACM will contain a sequence of data σ' such that one of the following is true:

1. $\sigma' = \sigma$: in this case neither the reader has removed any data item from the head of σ nor the writer has stored a new item in its tail;
2. $\sigma' = a_0 a_1 \cdots a_{j-1} a_j a_{j+1}$: in this case the reader has not removed any item from the head of σ, but the writer has added a new item to the tail;
3. $\sigma' = a_1 \cdots a_{j-1} a_j$: and, finally, in this case the reader has removed a data item from the head of σ.

The above can be specified by the following CTL formula:

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

where σ^+ is used to denote $a_0 a_1 \cdots a_{j-1} a_j$ or $a_0 a_1 \cdots a_{j-1} a_j a_{j+1}$ and σ^- is used to denote $a_1 \cdots a_{j-1} a_j$. 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.

The guidelines introduced above can be used to generate an SMV model for any RRBB ACM with three or more data cells. After that, the model can be verified against the CTL formulas for coherence and freshness. Observe that the number of CTL formulas needed to specify freshness grows linearly with the size of the ACM. This is because, for each possible size of σ, it is necessary to generate another CTL formula.

4 The implementation model and its verification

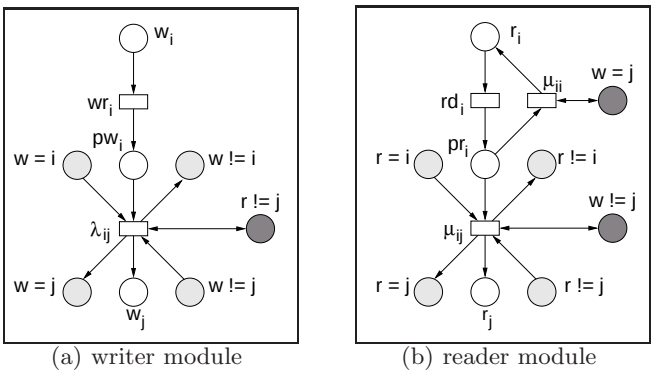
The modular approach for the generation of ACMs is now introduced by means of an example, including the generation of a Petri net implementation model and its verification.

4.1 Generation of the implementation model

A Petri net model for a 3-cell RRBB ACM will be generated and mapped into a C++ implementation. As stated before, this new modular approach is based on the definition of a set of elementary building blocks that can be easily assembled to construct the entire system.

The repetitive behavior of the writer consists of writing data into the i^{th} cell, checking if the reader process is addressing the next cell and, in the negative case advancing to it, otherwise waiting until the reader advances. In a similar way, the reader is expected to retrieve data from the i^{th} cell, check if the writer is accessing the next cell and, in the negative case advancing to it, otherwise preparing to re-read the contents of the i^{th} cell.

Two modules to control the access of each process to the i^{th} cell are defined. One corresponds to the behavior of the writer and the other to the behavior of the reader. The modules are shown in Figure 4.



(a) writer module (b) reader module

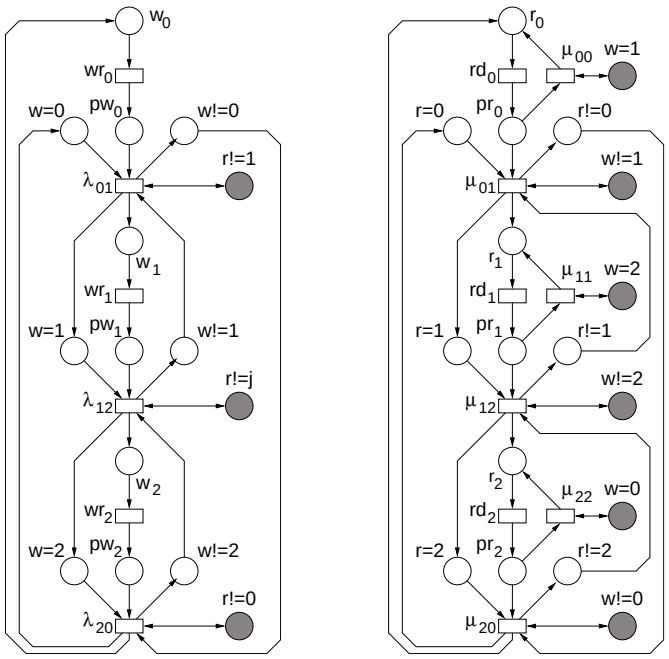
Fig. 4. Basic modules for the writer and the reader.

In Figure 4(a), a token in place w_i enables transition wr_i, that represents the action of the writer accessing the i^{th} cell. The places with label $\langle w = i \rangle$, $\langle w = j \rangle$, $\langle w \neq i \rangle$ and $\langle w \neq j \rangle$ indicate if the writer is pointing at the i^{th} or at the j^{th} cell. $\langle r \neq j \rangle$ indicates when the reader is pointing at the j^{th} cell. If transition λ_{ij} is enabled, then the reader is not pointing at cell j, the writer has just finished accessing the i^{th} cell and it can advance to the next one. The places $\langle w = i \rangle$, $\langle w = j \rangle$, $\langle w \neq i \rangle$ and $\langle w \neq j \rangle$ model the writer's control variables, and they are also used by the reader to control its own behavior. Note that $j = (i + 1) \mod n$.

The same reasoning used to describe the writer's module also applies to the reader's. The difference is that the reader should decide to advance to the next cell or to re-read the current cell. This is captured by the two transitions in conflict, μ_{ii} and μ_{ij}. Here the decision is based on the current status of the writer, i.e. if the writer is on the j^{th} cell or not, captured by a token on places $\langle w = j \rangle$ or $\langle w \neq j \rangle$ respectively. It is easy to realize that there is a place invariant

involving those places, since the sum of tokens is always equal to one, and only one of the transitions in conflict can be enabled at a time.

In order to create a process, it is only necessary to instantiate a number of modules, one for each cell, and connect them. Instantiating modules only requires replacing the i and j string by the correct cell numbers. For example, to instantiate the writer's module to control the access to the 0^{th} cell, the string i is replaced by 0 and j by 1. Connecting the modules requires to merge all the places with the same label. Figure 5 depicts the resulting Petri net models for the writer and reader of a 3-cell RRBB ACM.



(a) writer process (b) reader process

Fig. 5. The write and read processes for a 3-cell RRBB ACM.

After creating the processes, they can be connected by also merging places with same label on both sides. In this case, the shadowed places in each module will be connected to some place on the other module.

Definition 2 formally introduces the concept of a module. In this definition, it is possible to see that a module is an ordinary Petri net model that has some "special" places called ports. A port is a place that models a control variable. The *local ports* model the control variables that are updated by the process to which it belongs, while the *external ports* model the control variables updated by the other process. Ports are used to identify control variables when synthesizing the source code for an ACM.

Definition 2 (Petri net module) *A Petri net module is a tuple* $MODULE = (PN, LOC, EXT, T_a, T_c)$ *such that:*

1. PN *is a Petri net structure* (P, T, F) *with:*
   (a) P *being finite set of places.*
   (b) T *being finite set of transitions.*
   (c) $F \subseteq (P \times T) \bigcup (T \times P)$ *being a set of arcs (flow relation).*
2. $LOC \subset P$ *is a finite set of* local ports.
3. $EXT \subset P$ *is a finite set of* external ports *such that* $p \in EXT \iff p\bullet = \bullet p$. *Places in* EXT *are said to be read-only.*
4. $T_a \subset T$ *is a finite set of transitions such that* $t \in T_a \iff t$ *models a media access action.*
5. $T_c \subset T$ *is a finite set of transitions such that* $t \in T_c \iff t$ *models a control action.*
6. $T_a \bigcup T_c = T$ *and* $T_a \bigcap T_c = \emptyset$.
7. $M_a \subset (T_a \times \mathbb{N})$ *is a relation that maps each access transition* $t \in T_a$ *into an integer that is the number of the cell addressed by* t.
8. $M_c \subset (T_c \times \mathbb{N} \times \mathbb{N})$ *is a relation that maps each control transition* $t \in T_c$ *into a pair of integers modeling the current and the next cells pointed by the module.*

Definitions 3 and 4 formally introduce the writer and reader basic modules, respectively.

Definition 3 (RRBB writer module) *The RRBB writer module is a tuple* $WRITER = (PN_w, LOC_w, EXT_w)$ *where:*

1. PN_w *is as defined by Figure 4(a)*
2. $LOC_w = \{\langle w = i\rangle, \langle w = j\rangle, \langle w \neq i\rangle, \langle w \neq j\rangle\}$
3. $EXT_w = \{\langle r \neq j\rangle\}$
4. $T_a = \{wri\}$
5. $T_c = \{\lambda_{ij}\}$
6. $M_a = \{(wri, i)\}$
7. $M_c = \{(\lambda_{ij}, i, j)\}$

Definition 4 (RRBB reader module) *The RRBB reader module is a tuple* $READER = (PN_r, LOC_r, EXT_r)$ *where:*

1. PN_r *is as defined by Figure 4(b)*
2. $LOC_r = \{\langle r = i\rangle, \langle r = j\rangle, \langle r \neq i\rangle, \langle r \neq j\rangle\}$
3. $EXT_r = \{\langle w = j\rangle, \langle w \neq j\rangle\}$
4. $T_a = \{rdi\}$
5. $T_c = \{\mu_{ii}, \mu_{ij}\}$
6. $M_a = \{(rdi, i)\}$
7. $M_c = \{(\mu_{ii}, i, i), (\mu_{ij}, i, j)\}$

The connection of two modules, MOD_1 and MOD_2, is defined as another Petri net module that is constructed by the union of them. Definition 5 captures this.

Definition 5 (Connection for Petri net modules) *Given two Petri net modules MOD_1 and MOD_2, where:*

- $MOD_1 = (PN_1, LOC_1, EXT_1, T_{a_1}, T_{c_1}, M_{a_1}, M_{c_1})$ *and*
- $MOD_2 = (PN_2, LOC_2, EXT_2, T_{a_2}, T_{c_2}, M_{a_2}, M_{c_2})$.

The union of them is a Petri net module $m = (PN, LOC, EXT, T_a, T_c, M_a, M_c)$ such that:

1. $PN = PN_1 \bigcup PN_2$ *where* $P = P_1 \bigcup P_2$, *If two places have the same label them they are the same,* $T = T_1 \bigcup T_2$ *and* $F = F_1 \bigcup F_2$.
2. $LOC = LOC_1 \bigcup LOC_2$.
3. $EXT = EXT_1 \bigcup EXT_2$.
4. $T_a = T_{a_1} \bigcup T_{a_2}$.
5. $T_c = T_{c_1} \bigcup T_{c_2}$.
6. $M_a = M_{a_1} \bigcup M_{a_2}$.
7. $M_c = M_{c_1} \bigcup M_{c_2}$.

The complete ACM model can also be generated by the union of the Petri net models of each resulting process. The procedure is as introduced by Definition 5 except that rules 2 and 3 do not apply.

The last required step is to set an appropriated initial marking for the Petri net model. This can be done using Definition 6.

Definition 6 (Initial marking for RRBB ACMs) *For any Petri net model of an RRBB ACM, its initial marking is defined as follows. All the places are unmarked, except in these cases:*

1. $M_0(w_i) = 1$, *if* $i = 1$.
2. $M_0(\langle w = i \rangle) = 1$, *if* $i = 1$.
3. $M_0(\langle w \neq i \rangle) = 1$, *if* $i \neq 1$.
4. $M_0(r_i) = 1$, *if* $i = 0$.
5. $M_0(\langle r = i \rangle) = 1$, *if* $i = 0$.
6. $M_0(\langle r \neq i \rangle) = 1$, *if* $i \neq 0$.

Observe that according to Definition 6, the writer is pointing at the 1^{st} cell of the ACM and reader is pointing to the 0^{th} cell. By this, it can be deduced that the ACM is assumed to be initialized with some data on its 0^{th} cell.

4.2 Verification of the implementation model

The Petri net model generated using the procedure discussed above will be used to synthesize source code (C++, Java, Verilog, etc.) that implements the behavior specified by the model. So, it is necessary to guarantee that such a model is correct with respect to the behavior given by Definition 1 in Section 3. In this work, it is done by applying refinement verification. In other words it is

necessary to verify if the low-level specification, given by the Petri net model obtained as described above, implements correctly the abstract specification given by Definition 1.

Since Definition 1 was specified with the SMV language, it was necessary to translate the Petri net ACM model into SMV. The PEP tool [3] provides a way for translating a Petri net model into SMV and was used in our synthesis framework for such purpose.

The Petri net model specifies the mechanisms to control access to the ACM, but it does not model the data transfers. Since the goal is to check if the implementation model refines the abstract model, it is necessary to model data transfers in the implementation model. For that reason, a data array with the size of the ACM was added to the implementation model. For each event modeling a data access action, it was necessary to add the actions simulating the storage and retrieval of data in the array.

The following steps summarize what should be done to add the glue between the implementation and the abstract models.

1. Add a data array, with the same size as the ACM, to the SMV code of the Petri net model.
2. Identify in the SMV code generated by PEP the piece of code modeling the occurrence of each transition t of the Petri net model.
3. If t is a reader's action and $t \in T_a$, then the data stored in the i^{th}, where $(t, i) \in M_a$, position of the data array created in step 1 should be read.
4. If t is a writer's action and $t \in T_a$, then a new data item should be stored in the i^{th}, where $(t, i) \in M_a$, position of the data array created in step 1.

Note that the only control actions included in the model are required to avoid the non-determinism in the extra control variables. For instance, it is not desirable to allow non-deterministic changes in the values stored in the data array. By doing the above modifications in the SMV code of the generated Petri net model, it is possible to verify if the implementation model is a refinement of the abstract model with respect to the data read from the data array. It is important to note that the CTL formulae are defined in terms of the data array. Thus, if both models always read the same data from the array, and if the abstract model satisfies coherence and freshness, then the implementation model will also satisfy those properties and it can be used to synthesize the source code for the ACM.

Following the procedure described above a tool to automatically generate ACMs was designed and implemented[4]. A number of RRBBs with different sizes (starting from 3) where generated and proved to be correct for all cases.

[4] See `http://acmgen.sourceforge.net/` for details.

5 Synthesizing the source code

The implementation is generated from the Petri net model of each process. And the resulting source code is based on the simulation of the net model. So, the synthesis method consists of:

1. Create the shared memory as an array of the size of the desired ACM.
2. For each place p of the model, declare a Boolean variable vp named with the label of p and initialize it with the value of its initial marking. Note that if $p \in EXT$ then it will in practice be initialized by the other process, since in this case vp is seen as an external variable that belongs to another process.
3. For each transition t of the model, map into an **if** statement that is evaluated to true when all input variables of t are true. The body of the statement consists of switching the value of the input places of t to **false** and output places to **true**. If t models an access action, also add to the body of the **if** actions to write (or read) a new data item to (or from) the shared memory.

In order to perform the steps above, templates are used to define a basis for the source code of the ACM, then some gaps are fulfilled. More precisely, such gaps consist of: the declaration of the shared memory of a given size, the declarations of the control variable and the synthesis of the code that controls the access to the ACM.

Observe that the generation of the source code is performed from the Petri net model of each process and not from the model of the composed system. Algorithm 2 defines the basic procedure for the declaration and initialization of the control variables.

Algorithm 2 Control variables declaration and initialization

```
1: for all p ∈ P do
2:    if p ∈ LOC then
3:       Declare p as a local Boolean variable
4:       Initialize variable p with M0(p)
5:       Make variable p a shared one
6:    else if p ∈ EXT then
7:       Create a reference to a Boolean variable p that has been shared by the other
         process
8:    else
9:       Declare p as a local Boolean variable
10:      Initialize variable p with M0(p)
11:   end if
12: end for
```

In the first case, p is declared as a local Boolean variable that can be shared with the other processes and initialized with the initial marking of p. In the second case p is a shared Boolean variable that was declared in the other process

and in that case it cannot be initialized since it is a read-only control variable, from the point of view of the process being synthesized. Finally, in the third case, p is declared as a private Boolean variable and is initialized with the initial marking of p. In other words, each place will be implemented as a single bit unidirectional control variable. And each variable can be read by both processes but updated only by one of them.

Up to now the control part has not been synthesized, and there is no indication on how the data is passed from one side to the other. The shared memory can be declared statically as a shared memory segment and the only action needed to create it is to set the amount of memory that should be allocated to it.

Finally, the synthesis of the control for the reader and writer processes are introduced by Algorithms 3 and 4 respectively.

Algorithm 3 Synthesis of control for the reader

1: **for all** $t \in T$ **do**
2: **if** $t \in T_a$ with $(t, i) \in M_a$ **then**
3: Create new **if** statement
4: $\forall p \in {\bullet t}$ add to the **if** condition $p = true$
5: $\forall p \in {\bullet t}$ add to the **if** body $p = false$
6: $\forall p \in t{\bullet}$ add to the **if** body $p = true$
7: Add to the **if** body an instruction to read data from the i^{th} ACM cell
8: **else if** $t \in T_c$ with $(t, i, j) \in M_c$ **then**
9: Create new **if** statement
10: $\forall p \in {\bullet t}$ add to the **if** condition $p = true$
11: $\forall p \in {\bullet t}$ add to the **if** body $p = false$
12: $\forall p \in t{\bullet}$ add to the **if** body $p = true$
13: **end if**
14: **end for**

In Algorithm 3, the first case captures the synthesis of control to a data *read transition* addressing the i^{th} cell. The condition to the control is given by the pre-set of t and if it is satisfied then its pre-set it switched to $false$ and its post-set to $true$. And some data is read from the i^{th} cell. The second captures the synthesis of control to a *control transition*. As in the previous the condition is given by the pre-set of t and then its pre-set it switched to $false$ and its post-set to $true$.

Algorithm 4 is similar to Algorithm 3. The difference is that instead of reading some data from the i^{th} cell, the process will write some data into it.

The approach described here was used in the generation of C++ implementations for ACMs. In Figures 6 and 7 the methods that perform the shared memory accesses and control actions to the 3-cell RRBB ACM introduced in Section 4 are shown.

In Figure 6(a) it is possible to see the method that actually writes some data into the ACM. Line 3 captures the transition wr_0 in the Petri net model enabled. In this case: the variables implementing its pre-set are turned to `false`, line 4; the variables implementing its post-set are turned to `true`, line 5; and

Algorithm 4 Synthesis of control for the writer

1: **for all** $t \in T$ **do**
2: **if** $t \in T_a$ with $(t, i) \in M_a$ **then**
3: Create new **if** statement
4: $\forall p \in \bullet t$ add to the **if** condition $p = true$
5: $\forall p \in \bullet t$ add to the **if** body $p = false$
6: $\forall p \in t\bullet$ add to the **if** body $p = true$
7: Add to the **if** body a instruction to write new data on the i^{th} ACM cell
8: **else if** $t \in T_c$ with $(t, i, j) \in M_c$ **then**
9: Create new **if** statement
10: $\forall p \in \bullet t$ add to the **if** condition $p = true$
11: $\forall p \in \bullet t$ add to the **if** body $p = false$
12: $\forall p \in t\bullet$ add to the **if** body $p = true$
13: **end if**
14: **end for**

some data is written into the 0^{th} cell of the ACM, line 6. Note that the `val` is the new data to be sent and `shm_data` implements the shared memory. Note that each **if** statement refers to some transition in the Petri net model.

```
1.  void Writer::Send(acm_t val) {
2.
3.    if (w0 == true) { //wr0
4.      w0 = false;
5.      pw0 = true;
6.      *(shm_data + 0) = val;
7.    } else if (w1 == true) { //wr1
8.      w1 = false;
9.      pw1 = true;
10.     *(shm_data + 1) = val;
11.   } else if (w2 == true) { //wr2
12.     w2 = false;
13.     pw2 = true;
14.     *(shm_data + 2) = val;
15.   }
16. }
```

(a) Writer::Send()

```
1.  acm_t Reader::Receive(void) {
2.
3.    acm_t val;
4.
5.    if (r0 == true) {
6.      r0 = false;
7.      pr0 = true;
8.      val = *(shm_data + 0);
9.    } else if (r1 == true) {
10.     r1 = false;
11.     pr1 = true;
12.     val = *(shm_data + 1);
13.   } else if (r2 == true) {
14.     r2 = false;
15.     pr2 = true;
16.     val = *(shm_data + 2);
17.   }
18.
19.   return(val);
20. }
```

(b) Reader::Receive()

Fig. 6. Access actions implementation

The same reasoning applies to the reader access method shown in Figure 6(b). The only difference is that instead of writing into the ACM, it reads from there.

The methods implementing the control actions are somewhat more complex, but follow the same principle. The implementation of the writer's control actions are given by the method in Figure 7(a). As before, the same idea is used, implementing each control transition as an **if** statement whose condition is given by the variables of the pre-set and the body consists of switching the pre-set to `false` and the post-set to `true`. For example, the code implementing the firing

of transition λ_{01} is given by lines 3 to 14 of Figure 7(a). Observe that `we0` and `wne0` stands for $w = 0$ and $w \neq 0$ respectively.

The writer's control actions are inside an infinite loop whose last instruction is a call to a `pause()`[5] function. This is done because if there is no λ transition enabled, with the writer pointing at the i^{th} cell, it means that the reader is pointing at the $(i+1)^{th}$ cell. And in this case the writer should wait for the reader to execute. By using the `pause()` function in line 17, busy waiting algorithms are avoided. Also, note that the exit from the loop is done by a `break` statement, as in line 13.

```
1.  void Writer::Lambda(void) {
2.    while (true) {
3.      if (*we0 == true &&
4.          *wne1 == true &&
5.          w0p == true &&
6.          *rne1 == true) { // l0_1
7.        *we0 = false;
8.        *wne1 = false;
9.        w0p = false;
10.       w1 = true;
11.       *wne0 = true;
12.       *we1 = true;
13.       break;
14.     }
15.     if (...) {...} // l1_2
16.     if (...) {...} // l2_0
17.     pause();
18.   }
19. }
```

(a) Writer::Lambda()

```
1.  void Reader::Mu(void) {
2.    if (r0p == true &&
3.        *we1 == true) { // m0_0
4.      r0p = false;
5.      r0 = true;
6.      kill(pair_pid, SIGCONT);
7.    } else if (*re0 == true &&
8.        *rne1 == true &&
9.        r0p == true &&
10.       *wne1 == true) { // m0_1
11.     *re0 = false;
12.     *rne1 = false;
13.     r0p = false;
14.     r1 = true;
15.     *rne0 = true;
16.     *re1 = true;
17.     kill(pair_pid, SIGCONT);
18.   } else if (...) {... // m1_1
19.   } else if (...) {... // m1_2
20.   } else if (...) {... // m2_2
21.   } else if (...) {...} // m2_0
22. }
```

(b) Reader::Mu()

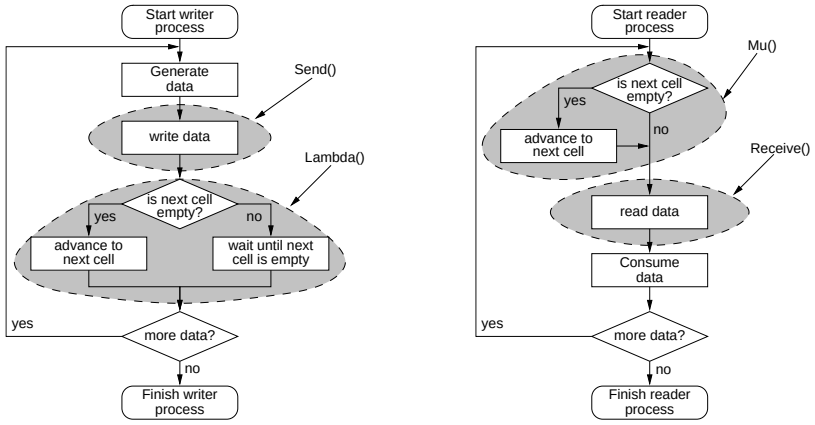
Fig. 7. Control actions implementation

The control actions of the reader process are implemented by the method in Figure 7(b). Again, each transition is implemented as an **if** statement. For instance, μ_{00} is implement by the code from line 2 to 6 and μ_{01} is implemented in lines 7 to 17. It is important to observe that every time the reader executes a control actions, it sends the signal `SIGCONT` to the writer, as in lines 6 and 17. This is to wake up the writer in the case it is sleeping due to a `pause()`.

Finally, the methods generated above need to be integrated into the communicating processes. As explained before and shown in Figure 8, the writer first calls the Send() and then the Lambda() methods. On the other hand, the reader first calls the Mu() and then the Receive() methods. In the code generated these operations are encapsulated into two public methods: Write() and Read(), available for the writer and reader respectively. With this, the correct use of the communication scheme is ensured.

In this Section an automatic approach to generate source code from Petri net models was discussed. The algorithms introduced here only gives conceptual

[5] The pause() library function causes the invoking process (or thread) to sleep until a signal is received that either terminates it or causes it to call a signal-catching function.



(a) Writer flow (b) Reader flow

Fig. 8. Flowchart for communicating processes

ideas on what needs to be done for the synthesis of the code. When executing the procedure, many details related to the target programming language has to be taken into account. The algorithms above were implemented to generate C++ code to be executed on a Linux system. The reader should consult [7] for more details on creating shared memory segments on UNIX systems.

6 Conclusions and future work

This work introduces a novel approach to the automatic synthesis of ACMs. The method presented here is based on the use of modules to the generation of Petri net models that can be verified against a more abstract specification.

Firstly, the behavior of RRBB ACMs was formally defined and the properties it should satisfy were described by CTL formulas. Then the procedure of generating the Petri net models was presented, including the definition of the basic modules and the algorithms required to instantiate and connect them. It was argued how the the resulting model is translated to an SMV model in order to be verified against the more abstract model defined in the beginning of the process. Finally, a C++ implementation is generated from the Petri net model.

Compared to the previous work [1], the method of generating Petri net models introduced here has the disadvantage of requiring model checking to guarantee its correctness. In the previous approach based on ACM regions, it was guaranteed by construction. However, the cost of executing the ACM regions algorithms is too high. And when it becomes limited by the state-space explosion problem, no implementation Petri net model could be generated and synthesis fails. In the approach proposed here, state-space explosion is limited to the verification of the Petri net implementation model. This step is off the design flow (see Figure 3). Thus we could generate C++ codes from the implementation model whether it

can be verified or not. An unverified implementation nonetheless has practical engineering significances because the Petri net model is highly regular and its behavior can be inferred from that of similar ACMs of smaller and verifiable size.

The next step into the direction of the automatic generation of ACMs is to provide a formal proof that the procedure of generating the net models is correct by design. With this, it will be possible to skip the verification step. And the time required to synthesize devices that can be trusted will drastically reduce. Also it is necessary to introduce formally the mechanisms used in the overwriting ACM classes. 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 has been supported by CICYT TIN2004-07925, a Distinction for Research from the Generalitat de Catalunya and by the EPSRC (grant EP/C512812/1) at the University of Newcastle upon Tyne.

References

1. Jordi Cortadella, Kyller Gorgônio, Fei Xia, and Alex Yakovlev. Automating synthesis of asynchronous communication mechanisms. In *Proceedings of the Fifth International Conference on Application of Concurrency to System Design (ACSD'05)*, pages 166–175, St. Malo (France), June 2005. IEEE Computer Society.
2. Jean-Philippe Fassino. *THINK : vers une architecture de systèmes flexibles*. PhD thesis, École Nationale Supérieure des Télécommunications, December 2001.
3. Bernd Grahlmann. The pep tool. In Orna Grumberg, editor, *Proceedings of CAV'97 (Computer Aided Verification)*, volume 1254 of *Lectures Notes in Computer Science*, pages 440–443. Springer, June 1997.
4. Leslie Lamport. On interprocess communication — parts I and II. *Distributed Computing*, 1(2):77–101, 1986.
5. Kenneth L. McMillan. *The SMV System: for SMV version 2.5.4*, November 2000. Available from: http://www-2.cs.cmu.edu/~modelcheck/smv/smvmanual.ps.gz.
6. Hugo R. Simpson. Protocols for process interaction. *IEE Proceedings on Computers and Digital Techniques*, 150(3):157–182, May 2003.
7. W. Richard Stevens. *Advanced programming in the UNIX environment*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.
8. 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.
9. 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.
10. Alex Yakovlev and Fei Xia. Towards synthesis of asynchronous communication algorithms. In Benoit Caillaud, Philippe Darondean, Luciano Lavagno, and Xiaolan Xie, editors, *Synthesis and Control of Discrete Event Systems. Part I: Decentralized Systems & Control*, pages 53–75. Kluwer Academic Publishers, Boston, January 2002.