# Towards the Automatic Synthesis of Asynchronous Communication Mechanisms

Kyller Costa Gorgônio
kyller@lsi.upc.edu

Advisor: Jordi Cortadella

*Draft submitted in Partial Fulfillment of the Requirements for the Degree of Doctor in Computer Science*

Universitat Politècnica de Catalunya
Departament de Llenguatges i Sistemas Informàtics
Barcelona, Spain – 2009

# Acknowledgments

I would like to thank my advisor Jordi Cortadella, who had had a lot of patience with me during the last years. He had guided me during this *journey*, always attempting to show me the way of the real and ethical research. I am glad I had the opportunity to work with him, and I hope to continue collaborating in the future.

I also would like to thank all my colleagues from UPC. I am quite sure that without them those years would have been very boring. Many thanks to all my friends in Brasil, from whom I robed some years. Special thanks to Angelo, Leandro and Gustavo who had give me their support on the most difficult times.

Many thanks to the Asynchronous Research Group at the University of Newcastle upon Tyne, in special to Alex Yakovlev and Fei Xia, for the discussions about the synthesis of asynchronous communication mechanisms.

This work was supported by the Ministry of Science, Culture and Sports of Spain, with the grant *"Beca de Postgrado para la Formación de Personal Investigador (FPI)"*, the project MAVERISH (*"Modelización, Análisis y Verificación de Sistemas Heterogéneos"*, TIC2001-2476-C03-01), the project GRAMMARS (Graph-based methods for the modelling, analysis and realization of large-scale systems, CICYT TIN2004-07925-C03-01), the project "Formal methods and algorithms for system design" (2007-2012, CICYT TIN2007-66523).

Finally, I would like to thank my family and to Michele (my eternal and beloved fiancée). They are the reason I am always trying to be a better person.

# Preface

The correct transfer of data between concurrent processes is of crucial importance in the exploitation of parallel architectures within distributed real-time systems. Techniques for solving this problem generally rely on mutual exclusion principles [61] to control access to shared communication resources. The traditional solution for mutual exclusion involves the use of semaphores, which can be easily configured to protect write and read operations on a shared memory in order to preserve the data being passed from one process to another.

The problem with this approach is that a minimum locking between the asynchronous communicating processes is not guaranteed. This is mainly because the semaphore protects the data access operations. Since these operations may be performed in a register of arbitrary size, this may take a long time to conclude. One way of solving this problem is to design the communication scheme in such a way that the atomic actions of each process only occurs at a very small granularity level, when accessing binary control variables.

An Asynchronous Communication Mechanism is a scheme which manages the transfer of data between two processes not necessarily synchronized for the purpose of data transfer. The general scheme of ACMs includes a shared memory for the data being transferred, and a set of unidirectional control variables. Each control variable is set by one process side and only read by the other. With this schema it is possible to design communication protocols in which the processes can be fully asynchronous. On the other hand, the use of binary control variables turns the design of ACMs a slow and prone to errors task.

This work describes an automated technique for the synthesis of ACMs. The construction of ACMs is known to be a hard task, and until now it was typically made on an ad-hoc basis, using a construct-and-verify approach. The main motivation of this thesis is the development of a systematic and automatic method for the construction of asynchronous communication mechanisms having as a starting point only its functional specification. At the end of the process, an implementation of the ACM will be available to be used to communicate between two asynchronous processes. Most important, the implementation obtained should have some guarantee of satisfying ACM properties such as coherence and freshness.

Firstly we will introduce ACMs, their classification, the properties they should satisfy, and their requirements. Two properties have special rele-

vance: *coherence* and *freshness*. Coherence is related to the mutual exclusion between the communicating processes when accessing the communication buffer. Freshness is related to the fact that a written data must be made available for the reader process. The requirements are that any control variable used must be single-bit and unidirectional, and the processes cannot share any action. The ACMs addressed here where introduced previously by Alex Yakovlev and Fei Xia. Unlike their work, which was focused on the design of ACMs, the main focus of ours was the development of methods to allow the rapid prototyping of ACM source code. In this sense, the work presented is an evolution of the original work by Yakovlev and Xia.

Two approaches, both having as input only a functional specification, for the automatic synthesis of ACMs will be described. The first one is based on the generation of a state graph specification for the ACM. The state graph model captures the properties of an ACM at the level of interleaving semantics. Then a Petri net model is synthesized from the state graph. The method for the synthesis of Petri nets is based on a more general procedure of synthesizing Petri nets, which uses the theory of regions. Finally, the implementation is derived from the Petri net model.

In this approach, the Petri net model synthesized preserves the coherence and freshness properties. This is guaranteed by construction. On the other hand, besides the fact that it is possible to generate state graph specifications of considerable size, the synthesis of Petri net models is only possible for ACMs of very small size, which makes the state graph based approach of limited practical use. From the analysis of the state graphs and the few Petri net models obtained, it was observed that both show a very regular structure. This suggested that it should be possible to explore this regularity to directly generate the Petri net model. Then, as before, the Petri net model can be used to obtain the ACM implementation.

Using this new approach, it is possible to generate ACMs of much larger size than before. The payback comes in the need to model check the Petri net model before synthesizing the implementation. In the first approach, the correctness of the PN was guaranteed by construction, which does not occur in the new one. For this reason, the formal verification of the ACMs models obtained with the second approach is addressed. Coherence and freshness are formally described as a set of CTL formulae. Then, model checking is applied to the Petri net model. If the result is positive, the ACM synthesis can process with some formal guarantee that the resulting artifact preserves coherence and freshness. On the other hand, since the state space of the ACMs may grow very fast, this new approach suffers from the state space explosion problem. It is not possible in practice to model check large ACMs. When it happens, it is still possible to synthesize the ACM implementation.

However, in this case it cannot be argued that coherence and freshness hold.

The synthesis of implementations of ACM from the Petri net models is discussed for both software and hardware. The software is synthesized as C++ source code. However, the technique presented is generic enough to be used to obtain source code for any programming language and OS which provides IPC and shared memory mechanisms. The synthesis procedure will be detailed later, and in short terms it is based on the simulation of the Petri net behavior by the source code.

Finally, the hardware implementation is given as a set of block diagrams and some Verilog code. The block diagrams implements the basic structure of the ACM, with some gaps that should be fulfilled. Those gaps are the behavior given by the writer and reader processes. The processes are implemented in Verilog code that is used to fulfill the gaps.

A number of applications can benefit from the synthesis of ACMs. In particular we can mention those in which a number of subsystems asynchronously communicate in order to achieve a common goal. For instance, let us consider an automotive embedded system in which many sub-modules monitor and/or control different parts of a vehicle. To control the speed of the vehicle a module can monitor the actual speed, another module may control the amount of gas injected into the engine, and yet another module may control the breaks. The interaction of all modules can be simplified if all sub-systems are asynchronous. If this happens, a proper communication scheme is needed. The scenario may become more complex when we add another sensor to check if there is some other vehicle on the way. In this case, the breaks should be activated and a new speed limit must be set. If the other vehicle starts running faster or changes the lane, then more gas may be injected into the engine to achieve the previous speed limit. The speed limiter system may also consider information about the legal speed limit on the road, which implies communication with some positioning system.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Chapter 1

# Introduction

The accurate and timely transfer of data between concurrent processes is of crucial importance in the exploitation of parallel architectures within distributed real-time data processing systems. Techniques for solving this problem generally rely on mutual exclusion principles [61] to control access to shared communication resources. The allocation of such resources is a classical example of the need for mutual exclusion in real systems.

Mutual exclusion is one of the most important mechanisms used for the synchronization of concurrent processes, and apart from its practical motivation, it is also a problem of great theoretical significance [44]. The mutual exclusion concept is strictly related to the way programmers think about concurrent programming. Therefore, a number of formal models of concurrency and the proposed interprocess communication mechanisms are based on the notion of mutual exclusion [35, 52].

The traditional solution for mutual exclusion involves the use of semaphores. A semaphore is a protected variable and constitutes the classic method for restricting access to shared resources in a multiprogramming environment [18]. The operations over a semaphore must be indivisible, which means that each of the operations may not be executed multiple times concurrently. A process wishing to execute an operation that is already being executed by another process must wait for it to complete first. This can be achieved by a special instruction, if the architecture's instruction set supports it, or by ignoring interruptions in order to prevent other processes from becoming active.

The use of semaphores provides solutions to most concurrent programming problems. However since the correct use of semaphores depends on all programmers involved in the construction of a system, it is difficult to construct a large system using them. If only one programmer makes a mistake when using the primitives, it will be very difficult to identify in which part

of the code the problem is [5].

To solve this problem, Hoare [34] and Hansen [30] proposed the use of *monitors* as a synchronization primitive that could be used in a more structured way. A monitor encapsulates all the procedures used to access a shared resource and all variables/data structures that describe its status. And the only way to access a resource is through the use of the monitor. Moreover, only one process can use a monitor at a time, this makes the monitors attractive to solve mutual exclusion problems.

Due to the fact that monitors can be embedded in programming languages, it is possible for the compiler to handle the mutual exclusion problem. This makes the use of monitors easier and safer than the use of semaphores, since mutual exclusion is guaranteed by the compiler and not by developers [71, 75].

## 1.1 Asynchronous communication mechanisms

Interprocess *Asynchrony* is inevitable for computation networks in the future. Firstly, this is because different and diverse functional elements, especially those connecting to analogue domains, tend to have different timing requirements [41, 69]. Secondly, concurrent and distributed system implementations lead to greater asynchrony between components as semiconductor technology advances and the degree of integration increases (the ITRS-2007 *"Design"* document [1] emphasizes multiple clock domains and source-synchronous signaling and predicts networks of self-timed blocks). The size of the computation networks is becoming larger, and the traffic between the processing elements is increasing. Therefore, handling the data communications which make up the traffic may determine much of the performance and characteristics of such systems.

One of the most important issues when designing communication schemes between asynchronous processes is to ensure that such schemes allow as much concurrency 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 to be achieved.

Classical semaphores can be easily configured to protect write and read operations on a shared memory in order to preserve the data being passed from one process to another. The communication model provided by semaphores is illustrated in Figure 1.1.

In this system a semaphore called *mutex*, that is properly initialized with value one before the initialization of the processes, is used to control the access to the shared memory. The **sender** produces the data that will be


Figure 1.1: Communication via shared memory

consumed by the **receiver**. Before writing or reading the memory, each process should perform a **Down** in the semaphore to guarantee that only one of them is executing the critical section.

```
1: process sender()
2:   loop
3:     produce data;
4:     Down(mutex);
5:     write on memory;
6:     Up(mutex);
7:   end loop
8: end process
```

```
1: process receiver()
2:   loop
3:     Down(mutex);
4:     read from memory;
5:     Up(mutex);
6:     process data;
7:   end loop
8: end process
```

However, this solution does not provide a minimum locking between the communicating processes, which is not satisfactory when it is desirable to increase the concurrency between them. This happens because the semaphore is directly protecting the data access operations, which may take a long time to conclude due to the fact that these operations may be performed on a register of arbitrary size.

One way to reduce the locking time between the processes is to design the communication scheme in such a way that the atomic actions of each process only occur at a very small granularity level, when accessing control variables. In other words, it is necessary to move the atomicity of actions from data accesses to few bits control variable accesses [69, 82]. Putting in a more simple way, what we want is that instead of entering into a critical section to perform an I/O operation, which may be long, the process only enters in the critical section to set the value of few and small control variables. And the value of these control variables will determine if the process has access guaranteed to some shared memory without the risk of other process accessing it at the same time.

The desire of reducing the locking time is exemplified by the original work of Lamport on *"atomic registers"* [45, 46] and it is present in all subsequent research on asynchronous communication, which includes the Lamport atomic register, in the literature. In Lamport's atomic registers, the communicating

processes are assumed to operate in fully temporal independence. However, the classification for asynchronous communication mechanisms proposed by Simpson [67, 70] introduced protocols that do not necessarily allow full asynchrony between the communicating processes. One of the processes, or both, may be required to wait for the other. In Simpson's work, the communication protocols are classified according to whether the reading and writing operations can be regarded as destructive or not. For instance, in a destructive writing scheme, the writer process cannot be blocked.

## The ACM model

An *Asynchronous Communication Mechanism* (ACM) is a scheme which manages the transfer of data between two processes, a *producer* and a *consumer*, not necessarily synchronized for the purpose of data transfer. In the context of ACMs, the producer and consumer of data are usually known as the *writer* and *reader* processes, respectively. Therefore, the ACM is a data connector linking two processes, the writer and the reader. The general scheme of these kinds of data communication mechanisms is shown in Figure 1.2. Most ACM implementations tend to include a shared memory, which is accessible to both processes, for the data being transferred, and a set of unidirectional control variables, each of which is set by one side and read by the other. In this work we assume that the data being transferred consists of a stream of items of the same type, and that the writer and reader processes consist of single-thread loops. During each cycle a single item of data is transferred to or from the ACM.

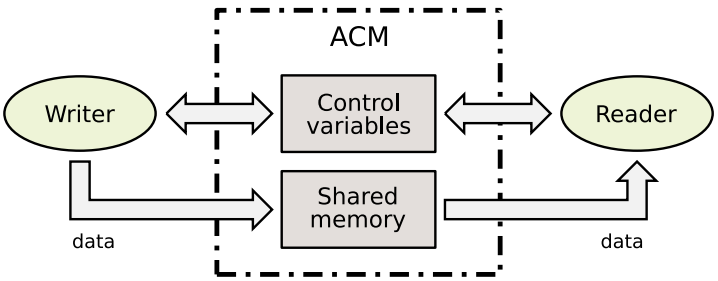



Figure 1.2: ACM with shared memory and control variables

ACMs may be of arbitrary size. The shared memory is organized as a ring of memory cells, as illustrated in Figure 1.3, each one being able to hold one data item. In some implementations that require overwriting it is necessary to have an extra space to store another data item in a cell, in this case we say that the cell has two slots. Each process attempting to access a certain cell

has the access to it granted or denied according to the values of the control variables and the requirements the ACM should satisfy.

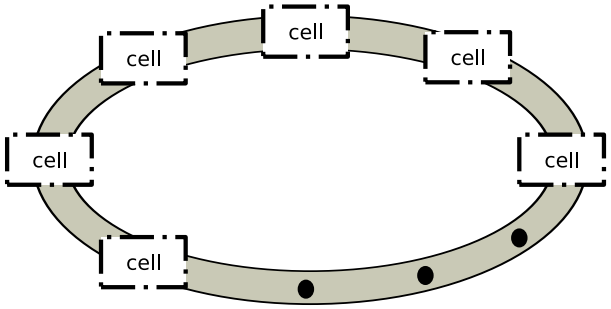



Figure 1.3: Multi-cell ACM scheme

Multi-cells ACMs are particularly important to give the designer more flexibility. In general, there is a trade off of data freshness for data sequencing. An ACM with more cells attempts to provide the reader process with better continuity of data. For instance, the user of a multimedia stream system cares more about receiving all frames of a video in the correct order than always getting the last rendered frame, even if few frames are lost. On the other hand ACMs with few cells attempt to serve the reader with fresh data. In a system that controls the temperature of a room, probably the last data received from the sensors are more important than any previous data. Depending on the requirements of the system under development, the designer determines the optimum size of the ACM.

ACMs are classified according to whether overwriting and re-reading are permitted [86].

- **Overwriting**: occurs when the ACM is full of data that has not been read yet, in this case the writer can discard the oldest data item and overwrite it;
- **Re-reading**: occurs when all the data in the ACM has been already read, and in this case the reader is allowed to read the last accessed item again.

Table 1.1 shows such a classification. BB stands for a bounded buffer and it does not allow overwriting or re-reading. Depending on the circumstances, it may be necessary for one of the processes to wait for the other. The RRBB ACM specifies a communication scheme that permits re-reading, but the writer is required to wait if the buffer is full of data that has not been read. On the other hand the OWBB scheme allows overwriting items but

requires the reader to wait for new data if the buffer is *empty*. Finally, the OWRRBB scheme does not require either process to wait at any time. This simple and elegant classification is used on the work presented here.

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

Table 1.1: ACMs classification.

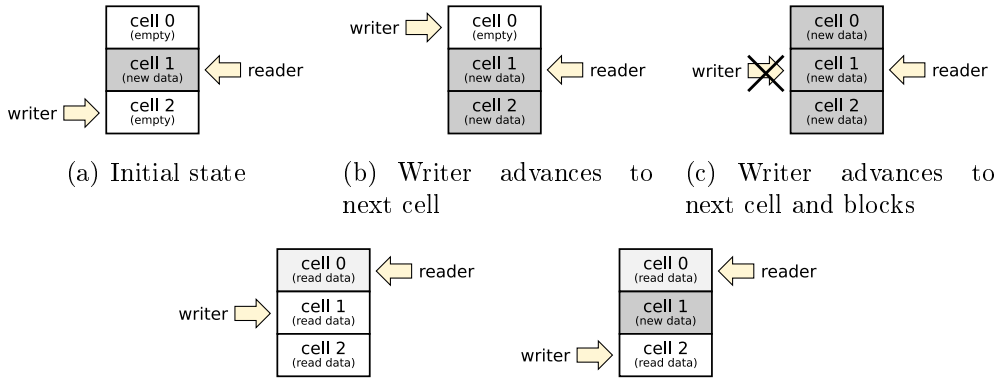
For re-reading, it is much more natural to re-read the item from the previous cycle rather than re-reading an item from several cycles before. On the other hand, overwriting might happen anywhere in the buffer. However, the strongest practical cases that have been proposed consist of overwriting either the newest or the oldest item in the buffer [22, 69, 84]. Overwriting the newest item in the buffer, which is a relatively straightforward task to approximate [84], 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. Note that in this case, only the oldest data item is lost, the others remain available for reading. In this work, we tackle the much more interesting problem of overwriting the oldest item in the buffer.

## An introductory example

To illustrate how an ACM works, let us consider a re-reading (RRBB) ACM with three data cells. The control variables **r** and **w** are used to indicate which cell the reader and the writer should access, respectively. In the initial state of the graph in Figure 1.4(a), the reader is pointing at cell 1, which is also initialized with some data, and the writer to cell 2 (r = w - 1).

The behavior of the writer is as follows. It first writes the data on the $w^{th}$ cell, then it advances to the next cell by increasing (in a modulo operation) the value of **w**. Finally, it checks if the reader is also pointing at the new cell, i.e. if **w** is equal to **r**. In the positive case, since overwriting data is not allowed, the writer waits until it is no longer true. Otherwise it can access the ACM again, as illustrated by Figures 1.4(c) and 1.4(b) respectively.

The reader first checks if the writer is pointing at the next cell, i.e. if $(r + 1) \mod 3 \neq w$. In the positive case, the reader prepares to re-read the data on the current cell and the value of $r$ does not change, as in Figure 1.4(d).




(a) Initial state

(b) Writer advances to next cell

(c) Writer advances to next cell and blocks

(d) Reader prepared to re-read

(e) Reader prepared to advance to next cell

Figure 1.4: Execution of RRBB ACM with 3 cells.

Otherwise it advances to the next cell by incrementing $r$ and prepares to read a new data item, as in Figure 1.4(e).

By behaving like this, 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 in case there is no unread data in the ACM.

**Algorithm 1.1** Re-reading ACM with 3 cells

```
1: process writer()
2:    w := 2;
3:    loop
4:       write cell w;
5:       w := (w+1)  mod  3;
6:       wait until w ≠ r;
7:    end loop
8: end process
```

```
1: process reader()
2:    r := 1;
3:    loop
4:       if (r+1)  mod  3 ≠ w then
5:          r := (r+1)  mod  3;
6:       end if
7:       read cell r;
8:    end loop
9: end process
```

The behavior of the reader and the writer is captured by Algorithm 1.1. Observe that each control variable is updated by only one of the process. By doing so, unidirectional control variables as introduced in the previous section are used.

In Algorithm 1.1 it is necessary two bits to represent each control variable. If one variable is about to be modified and referenced about the same time, it is not possible to guarantee the consistence of the recovered value. This

means that it is necessary to encode each **r** and **w** as a set of three binary variables, $w_i$ and $r_i$, with $i = 0, 1, 2$. $w_i$ will have value 1 if the writer is pointing to the $i^{th}$ cell, and the values of the $w_i$ changes according to the cell that the writer is pointing to. The same reasoning applies to the reader. It is not difficult to see that when the size of the ACM grows, it becomes more and more difficult for a human developer to deal correctly with all control variables. This becomes even more evident for the overwriting ACM policies.

Besides that, when designing ACMs it is necessary to consider that the ACM should satisfy some data coherence and freshness requirements. The data coherence requirement is related to the fact that the reader and the writer processes cannot access the same data record at the same time to avoid the reader obtaining a data that is not valid, and the freshness requirement is related to the fact that the last data record produced by the writer process must be made available to the reader.

## 1.2 Automatic synthesis of ACMs

The main motivation of this thesis is the proposal of a formal and automatic method for the construction of asynchronous communication mechanisms having as a starting point only its functional specification. At the end of the process, an implementation of the ACM will be available to be used for communication between two asynchronous processes. Two types of implementations will be addressed, software and hardware implementations. Each type of implementation has its own technicalities that will be addressed at the proper time. For now it is enough to know that in the end of the process, a C++ or a Verilog implementation will be made available for use.

The main characteristic of the artifacts we want to synthesize is that they must provide communication using the model described in the previous sections, which is illustrated in Figure 1.2. The access to the shared memory must be controlled through the use of a set of binary and unidirectional control variables.

In order to achieve the goal declared above, two methods are proposed. The first method is based on the generation of a state graph specification from the ACM functional specification [16, 26]. The state graph model captures the properties of an ACM at the level of interleaving semantics. Then a Petri net [55, 57] model is synthesized from the state graph. The method for the synthesis of Petri nets is based on a more general procedure of synthesizing Petri nets, which uses the theory of regions [56]. Finally, the implementation is derived from the Petri net model. This method is illustrated in Figures 1.5.

This was our first effort in order to automate the task of generating ACMs.

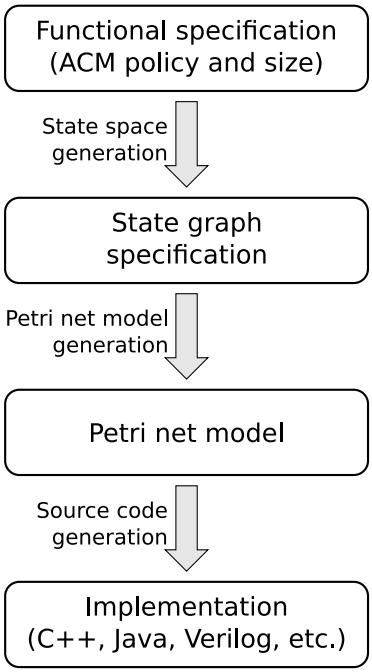



Figure 1.5: Design flow for ACMs.

This technique was developed under the assumption that it is more natural to generate code from the Petri net model if this model can be decomposed into two sub-models, sharing some few places, which model the behavior of two sequential processes that are truly concurrent. Considering this, a Petri net model is closer to the implementation than state graphs.

This approach has the advantage that it is guaranteed by construction that the Petri net model synthesized preserves all properties a correctly designed ACM should satisfy. On the other hand, as nothing comes for free, this advantage comes at a high price. Besides the fact that it is possible to generate state graph specifications of considerable size, the synthesis of Petri net models is only possible for ACMs of very small size. This makes the state graph based approach of limited practical use. It is clear that an alternative method for the automatic synthesis of ACMs is needed in practice.

From the analysis of the state graphs and the few Petri net models obtained with the application of the state graph approach, it was observed that both show a very regular structure. This suggests that it should be possible to explore this structured nature and define a new method that directly generates the Petri net model, avoiding the need of synthesizing it from a state graph specification [25]. Then, as before, the Petri net model can be used

to obtain the ACM implementation. This new method for the automatic generation of ACMs is illustrated in Figure 1.6.

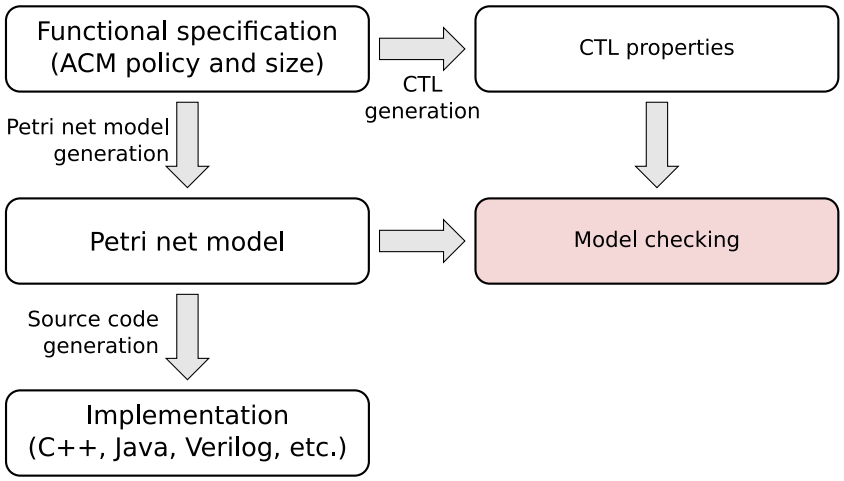



Figure 1.6: Design flow for ACMs. Optimized approach.

Using this new approach, it was possible to generate ACMs of much bigger size than before. The payback comes in the need to model check the Petri net model before synthesizing the implementation. In the first approach, the correctness of the PN was guaranteed by construction, which does not occur in the new one. For this reason, it is necessary to provide some formal guarantee of such correctness. In order to address this problem, coherence and freshness, have been formally described as a set of CTL formulae [12]. Then, models checking can be applied over the Petri net model. If the result is positive, the ACM synthesis can process with some formal guarantee that the resulting artifact preserves coherence and freshness.

On the other hand, since the state space of the ACMs may grow very fast, this new approach suffers from the state space explosion problem. It is not possible to model check large ACMs. When it happens, it is still possible to synthesize the ACM implementation. However, in this case the correctness of coherence and freshness cannot be guaranteed.

It is interesting to observe that in Figure 1.6 it is not specified what happens if Model Checking fails. There is no feedback to previous steps if something goes wrong. This is because the Model Checking is used to validate our synthesis methodology. It is not intended to be used by the *user*. This was necessary due to the fact that we do not have any formal proof of its correctness.

The Petri net model can be used to synthesized both software [25] and

hardware [24] implementations. The synthesis procedure will be detailed later, and it is based on the simulation of the Petri net behavior by the source code.

## 1.3 Organization of the document

The rest of this document is organized as follows. In Chapter 2 some basic definitions and the state of the art in the specification of asynchronous communication mechanisms are given. Once the background is introduced, the main topics will be addressed.

In Chapter 3 the first method for the automatic synthesis of ACMs is introduced. The generation of state graph specification for re-reading and overwriting ACMs is detailed as is the generation of Petri nets from those specifications. The Petri net approach, which does not depend on the state graph specification, is detailed in Chapter 4. This second approach is based on the definitions of modules that are used as building blocks to generate the PN models. Also, the generation of PN models for re-reading and overwriting policies is presented in details. Besides that, the CTL formulae for coherence and freshness properties are introduced.

Then the automatic formal verification of the PN models is discussed in Chapter 5. The definitions presented in Chapter 4 are used in order to define a verification methodology that uses an abstraction of the ACMs for checking coherence and freshness. Then, refinement verification is applied to ensure that the PN model correctly implements the ACMs abstraction with respect to those properties.

In Chapters 6 and 7 the realization of the PN models as real implementations is discussed. Firstly, in Chapter 6, the generation of software implementations is presented. The ACM is implemented as C++ source code design to run in a specific environment. The same technique can also be used to obtain implementation in other programming languages and/or operating systems, provided that the needed inter-process communication primitives are provided. Secondly, in Chapter 7, the generation of hardware implementations for the re-reading ACM is presented. The implementation is given in the form of Verilog code and as a set of block diagrams, which can be easily mapped to Verilog code.

# Chapter 2

# Preliminaries

The basic concepts and definitions required to the development of this work is introduced here. In Section 2.1 the Petri nets, which will be used to specify the ACM model, are formally described. Then, in Section 2.2, the model checking verification technique is discussed. Also, the CTL temporal logic is introduced. The CTL model checking will be used later to formally verify coherence and freshness over the obtained ACM models. Section 2.3 brings a discussion about Lamport's atomic registers, which have inspired all work on ACMs. In Section 2.4 Simpson's four-slots, which addresses policies not considered by Lamport, are introduced. Finally, in Section 2.5 the systematic approach for the generation of ACMs given by Xia and Yakovlev is discussed. Besides that, the ACMs requirements proposed by them are detailed. These requirements are important to us due to the fact they have been considered in the design of the automatic ACM generation methods described later.

## 2.1 Petri nets

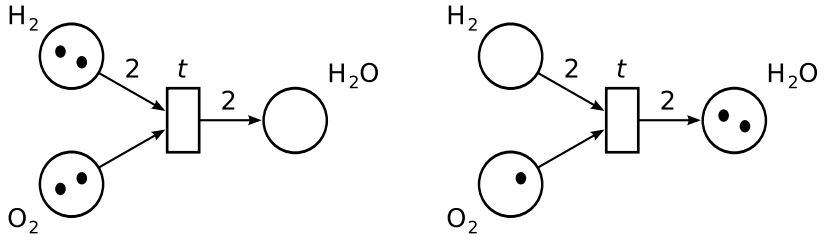
Petri nets (PN) are a mathematical tool that can be used in the modeling and analysis of systems, especially concurrent, asynchronous, distributed and non-deterministic ones [55]. They have a graphical representation that is equivalent to its mathematical notation. This graphical representation turns the comprehension of the system being modeled easier.

A Petri net is composed of a net structure and an initial marking associated to this structure. The Petri net structure is given by a bipartite graph in which the nodes are of two types: *places* and *transitions*. Places usually represents states or resources of the system, while transitions usually model events or actions. A Petri net is formally described in Definition 2.1

**Definition 2.1 (Petri net)** *A Petri net is a tuple* $N = \langle P, T, F, W, M_0 \rangle$ *in which:*

- $P$ *is a finite set of places;*
- $T$ *is a finite set of transition;*
- $P \bigcap T = \emptyset \wedge P \bigcup T \neq \emptyset$*;*
- $F \subseteq (P \times T) \bigcup (T \times P)$ *is a set of arcs (flow relation);*
- $W : F \rightarrow \mathbb{N} - 0$ *is a weight function;*
- $M_0 : P \rightarrow \mathbb{N}$ *is the initial marking.*

According to the relation flow $F$, the arcs in the Petri net always connect nodes of different types. i.e. a place to a transition, or a transition to a place. In the graphical notation, places are represented by circles, transitions by rectangles and the relation flow by directed arcs. This graphical representation is illustrated in Figure 2.1. This Petri net, models the chemical reaction to obtain one molecule of water, $H_2O$.




(a) Before the reaction (b) After the reaction

Figure 2.1: PN introductory example, a chemical reaction

The state of a Petri net is given by its marking. A marking corresponds to the number of tokens associated to each place at a given moment. For instance, a token on place $H_2$ indicates the presence of one molecule of hydrogen in the system. Specifically, on Figure 2.1(a) there are two molecules of hydrogen and two of oxygen to perform the chemical reaction to obtain water. This reaction is modeled by transition $t$. When the reaction happens, two molecules of hydrogen and one of oxygen are used to generate two molecules of water, and the state of the Petri net changes to the one shown in Figure 2.1(b).

The semantics of a Petri net is defined by the firing rule, which establishes the conditions that should be satisfied to enable the occurrence of a transition

and what are the consequences of *executing* that transition. For instance, in the Petri net of Figure 2.1, the condition to enable $t$ is the existence of two token on place $H_2$ and one token on place $O_2$. The firing rule states that:

1. A transition $t$ is enabled if each of its input places has at least the amount of tokens specified in the arc from the respective input place;

2. An enabled transition may or may not be fired;

3. When a transition is fired, the number of tokens specified on the input arcs are removed from the respective places and tokens are added to all output places according to the weight of the respective arcs.

In the system above, when the transition $t$ is fired two tokens are removed from $H_2$ and one token is removed from $O_2$, then two tokens are added to $H_2O$. The notation $\bullet t$ and $t\bullet$ is used to denote the pre-set and the post-set of $t$, i.e. the set of input places and the set of output places of $t$, respectively. The same notation is used to denote the pre-set ($\bullet p$) and post-set ($p\bullet$) of a given place $p$.

Let us consider a transition $t_i$ that fires on marking $M_i$ and results in marking $M_j$, which is denoted by $M_i[t_i > M_j$. Lets also consider a transition $t_j$ that fires on $M_j$ and results in $M_k$, which is denoted by $M_j[t_j > M_k$. $M_i[\sigma > M_k$, where $\sigma = t_i t_j$, can be used to say that $M_k$ can be reached from $M_i$ by the firing sequence $\sigma$. In particular, a marking $M$ is said to be reachable from the initial marking $M_0$ of a given Petri net if there exists a firing sequence $\sigma$ such that $M_0[\sigma > M$.

The Petri net can be used to analyze two types of properties: *behavioral* and *structural*. The behavioral properties are those that depend on the initial marking of the Petri net, while the structural properties do not. Typical properties that can be analyzed include *reachability*, *boundness*, *liveness*, *reversibility*, *coverability* and *persistence*. Boundness property is of particular interest in this work. A Petri net is said to be *k-bounded* if the number of tokens in each place does not exceed a finite number $k$ for any reachable marking of the Petri net for a given $M_0$. In particular, the Petri net is said to be one-safe if $k = 1$. A marking $M$ is said to be reachable from $M_0$ if there is a firing sequence that changes marking from $M_0$ to $M$.

## 2.2 Model checking

Formal verification is a systematic approach that helps designers to reason about the behavior of computational systems. Models and properties (also

called specification) are described using languages with mathematical semantics, and they can be used to identify design errors and to prove that a given model satisfies, or implements, its specification [15].

Model checking is a verification technique in which the system to be verified is described as a state transition system with finite behavior and the properties of the system are described as temporal logic formulae [13]. The goal is to determine if a temporal logic formula $f$ is true in a given transition system. In short terms, the verification is based upon an exhaustive search of the state space of the system.

Formally, the model checking problem is defined in terms of a Kripke structure. A Kripke structure is a non-deterministic finite state machine whose states are labeled with Boolean variables, which are the evaluations of expressions in that state [14, 36]. The model checking problem can be stated as follows: Let $K$ be a Kripke structure, and let $\varphi$ be a temporal logic formula. Find all states $s$ of $K$ such that $K, s \models \varphi$ [8].

Temporal logic is a modal logic that consists of a formal framework that could be used to describe how the events of a given system occur in the time [21]. Typically, the operators of the logic allow describing safety, liveness or precedence properties. In this way, temporal logic provides a useful framework to specify software systems, especially concurrent systems, as proposed by Pnueli in [58].

## CTL

Among others, the Computation Tree Logic (CTL) and Linear Temporal Logic (LTL) are two of the most useful temporal logics [9, 10]. They differ in how they handle branching in the underlying computation tree. The CTL operators permit to quantify over the paths departing from a given state. In LTL, operators are intended to describe properties of all possible computation paths. It is an agreement that the temporal logic provides a good framework to describe and to reason about the behavior of concurrent systems. However, it is not the case when the question is which one is more appropriate, linear or branching time logic, to do it. In this work the Cadence SMV [50] model checker will be used for the verification of ACM models, for this reason we will concentrate on the description of the CTL temporal logic only.

CTL combines path quantifiers with linear time temporal logic operators. The path quantifiers **A** (*"for all paths"*) and **E** (*"for some paths"*) should be used as a prefix of one of the operators **G** (*"always"*), **F** (*"sometimes"*), **X** (*"nexttime"*) and **U** (*"until"*).

Given that $AP$ is the set of atomic propositions, the syntax of CTL is given by the following rules:

1. If $\phi \in AP$, then $\phi$ is a CTL formula;
2. If $\phi$ and $\varphi$ are formulae, then $\neg\phi$, $\phi \wedge \varphi$ and $\phi \vee \varphi$ are also formulae;
3. If $\phi$ and $\varphi$ are formulae, then $\mathbf{EX}(\phi)$, $\mathbf{AX}(\phi)$, $\phi$ $\mathbf{EU}$ $\varphi$ and $\phi$ $\mathbf{AU}$ $\varphi$ are also formulae.

The others CTL operators are defined as abbreviations as follows:

$$\begin{aligned} \mathbf{EF}\ \phi &\equiv \texttt{true}\ \mathbf{EU}\ \phi \\ \mathbf{AG}\ \phi &\equiv \neg\mathbf{EF}(\neg\phi) \\ \mathbf{AF}\ \phi &\equiv \texttt{true}\ \mathbf{AU}\ \phi \\ \mathbf{EG}\ \phi &\equiv \neg\mathbf{AF}(\neg\phi) \end{aligned}$$

A CTL formula is interpreted with respect to a Kripke structure. If $\phi$ is a CTL formula, $K, s_0 \models \phi$ is used to denote that $\phi$ holds in the state $s_0$ of $K$.

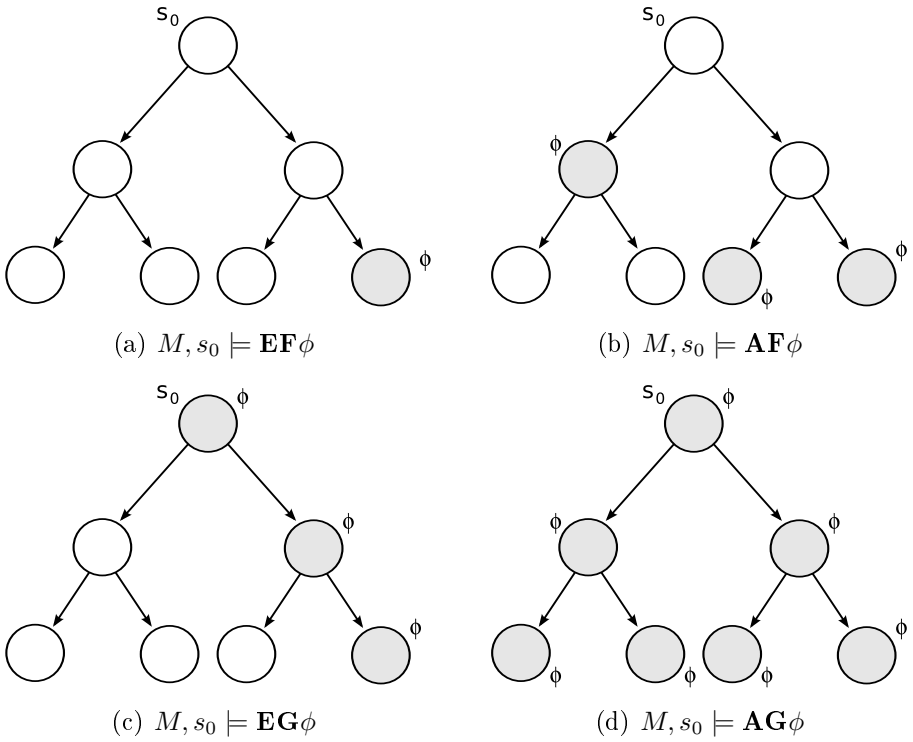
The four most used CTL operators are **EF**, **AF**, **EG**, and **AG**. The intuitive interpretation of such operators is illustrated in Figure 2.2. $\mathbf{AF}(\phi)$ means that for all paths starting from $s_0$, $\phi$ holds at some state along the path, $\phi$ is said to be *inevitable*. $\mathbf{EF}(\phi)$ means that exists a path starting from $s_0$ in which $\phi$ holds at some state along this path. $\mathbf{AG}(\phi)$ means that for all paths starting from $s_0$, $\phi$ holds at every state along those paths, i.e. $\phi$ holds *globally*. Finally, $\mathbf{EG}(\phi)$ means that exists a path starting from $s_0$ in which $\phi$ holds at every state along this path.

## 2.3 Lamport's *"Atomic Registers"*

In [46] Lamport showed that the problem of asynchronous interprocess communication can be solved through the use of shared registers. He also introduced a set of algorithms to do this. In his work, Lamport defined three classes of registers: *safe*, *regular*, and *Atomic*.

The *safe* register is the weakest one. It is necessary to assume only that reads and writes cannot be concurrent. If this requirement is satisfied, then the reader will obtain the most recently written value. Otherwise, no assumption is made and the reader can obtain any of the possible values of the register.

The *regular* register is a *safe* register in which a read may overlap a write. If it happens, then the reader will receive either the new value of the register or its previous value. Note that if the read overlaps a series of writes, the




(a) $M, s_0 \models \mathbf{EF}\phi$ (b) $M, s_0 \models \mathbf{AF}\phi$

(c) $M, s_0 \models \mathbf{EG}\phi$ (d) $M, s_0 \models \mathbf{AG}\phi$

Figure 2.2: Basic CTL operators

value obtained by the reader will be either the value in the register before the read started or any of the values written during some of the overlapping writes.

The *atomic* register is the strongest possibility. It is a *safe* register in which for any execution of the system there is some sequential execution order of the reads and writes such that the values obtained by the reads are the same as if the operations had been executed in such order.

To better understand these registers, let us consider a system composed of two concurrent process (one writer and one reader) that access one shared register that can have values X, Y or Z. In Figure 2.3 a possible execution of the system is presented. Each operation, is represented by a horizontal *temporal* line, wr stands for "write" and rd stands for "read".

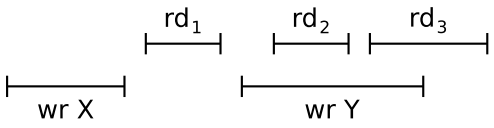



Figure 2.3: Two writes and three reads execution

If the register of the system is *safe* then the operation $rd_1$ will obtain X as the result. On the other hand, $rd_2$ and $rd_3$ may obtain X, Y or Z (all the possible values of the register) as the result.

In the case of a *regular* register $rd_1$ also obtains a X, but now $rd_2$ and $rd_3$ cannot obtain Z as a result since it was neither the previous value of the register nor the new value. In particular, it is important to note, $rd_2$ can obtain Y while $rd_3$ obtains X.

Finally, in the case of an *atomic* register $rd_1$ obtains X while $rd_2$ and $rd_3$ should obtain respectively: X and X; or X and Y; or Y and Y. Note that now $rd_3$ cannot obtain X if $rd_2$ obtained Y before. It happens because for any possible sequential execution of this system, $rd_2$ must necessarily occurs before $rd_3$, and if the value obtained by $rd_2$ is Y, it means that in the sequential execution wr Y has finished and since the register is also *safe* the only possible value $rd_3$ can obtain is Y.

According to Lamport, the implementation of a reader-writer register should consider three main dimensions:

1. *safe* / *regular* / *atomic*;
2. *Boolean* / *multivalued*;
3. *one-reader* / *multireader*.

The first dimension is defined according to the assumptions that the register must satisfy, i.e. if the register is *safe*, *regular* or *atomic*. The second dimension is related to the values that a register may assume, i.e. if the register can assume only Boolean values or if it can assume any value of a specified set. Finally, the third dimension classifies the register according to the number of reader processes allowed to access the registers.

Together, these three dimensions generate twelve different classes of implementations that can be partially ordered by its strength. For example, an atomic, multivalued, multireader register is stronger than a safe, Boolean, one-reader register (which is the weakest possibility). Lamport defined methods to construct a register of each class using only registers of a weaker class. For instance, a multireader, Boolean, safe register is constructed from one-reader registers. The method consists in replicating the one-reader register in such a way that each register will be accessed by only one reader and one writer will provide data to all of them.

Although Lamport claims the weakest register is the safe, Boolean and one-reader, it is necessary to observe that the read of a persistent register that overlaps with a write cannot guarantee to return a valid value (if the ACM is physically able to store more values than it is specified by the data

type) [33]. For example, to implement a 3-valued register we need at least two bits, and thus the register is able to store four different values.

## 2.4 Simpson's Four-slot ACM

Simpson defined a fully asynchronous communication mechanism that makes use of a shared memory to communicate data between two processes [65]. Mutual exclusion is replaced by a control scheme that avoids any concurrent read and write to the same data slot (a slot is an area in the shared memory that is used to store one data record). Simpson's ACM requires the use of four slots and four binary control variables to guarantee that the reader and writer processes will not access the same data slot at the same time to preserve data coherence.

The four slots are organized into two pairs, and there is an initialization procedure that puts one data record in one of the slots. The four binary control variables are used to indicate which slot each process is accessing. So we have the following variables:

1. *reading*: indicates the pair of slots the reader is accessing, or last accessed;

2. *latest*: indicates the pair of slots the writer is accessing, or last accessed;

3. *slots*: it is a two-element array of binary variables that are used to indicate the slot number (zero or one) each process should access in the pair it is currently pointing to.

Essentially, the writer and reader processes behave as described in Algorithm 2.1.

The writer algorithm always avoids selecting the pair the reader is currently pointing to, and avoids selecting the slot in the pair which was last written. With this correct choice, the process can proceed writing to the memory and then indicating the pair and slot that contains the latest written data element.

The reader algorithm always chooses the pair and slot that contains the latest written data, and then indicating that it is pointing to this memory area. Then the reader can access the new data element.

The design of this four-slot mechanism intends that the reader and writer never access the same slot at the same time. This guarantees that data coherence is preserved even when reads and writes overlap. In the same way, since the reader always accesses the most recently written item, it also preserves data freshness. More specifically the reader:

**Algorithm 2.1** Simpson's four-slot ACM

```
1:  process writer()
2:    var pair, index: boolean;
3:    loop
4:      pair = ¬ reading;
5:      index = ¬ slots[pair];
6:      write new data record to slot
        (pair,index);
7:      slots[pair] = index;
8:      latest = pair;
9:    end loop
10: end process
```

```
1: process reader()
2:   var pair, index: boolean;
3:   loop
4:     pair = latest;
5:     reading = pair
6:     index = slots[pair];
7:     read data record from slot
       (pair,index);
8:   end loop
9: end process
```

1. Obtains the last data record written before the start of the read, when the read does not overlap with a write;

2. Obtains the last data record written before the start of the read or one of the data records written by an overlapping write;

3. Never obtains a data record that is older than the previously obtained one.

Observe that these relate directly to the *safe*, *regular* and *atomic* registers defined by Lamport [46].

Later Simpson presented an analysis method [66, 68] and used this method to demonstrate that data coherence and freshness are guaranteed by the four-slot ACM.

## 2.5 A systematic approach to synthesis of ACMs

The construction of algorithmic descriptions that can be mapped into hardware or software implementations is a challenging task. A number of solutions have been proposed. However, most of them have been obtained through an ad hoc process. Xia and Yakovlev [78, 81, 85] introduced a systematic method to synthesis of algorithmic descriptions of ACMs. Their method consists in four steps:

1. Construction of the state graph specification of the ACM;

2. Refining the state graph through the addition of silent actions that are interpreted as an update in the value of a set of control variables;

3. Synthesizing a Petri net model that is composed by two sequential processes from the refined state graph;

4. Mapping the obtained Petri net model into an algorithmic description.

The two main assumptions of that work are that: i) the communicating processes do not share any action in order to transmit data; and ii) all control variables used are unidirectional. In what follows, these two assumptions are detailed.

### 2.5.1 Communication without process sharing

Interprocess data transmission is traditionally implemented through synchronization for the purpose of data transfer (e.g. Occam [40]). However, this kind of synchronization, especially when including actions which may be regarded as simultaneously belonging to both communicating processes ("process sharing"), is undesirable for ACM solutions unless demanded by the functional specification. In [79, 80, 82, 83, 85, 86] much effort has been spent on making ACMs as concurrent as possible, given a particular specification. This work continues in that direction.

Functional specifications of ACMs may require synchronization between the processes at some point. For instance, let us consider a simple ACM, called "rendezvous" in [69]. This ACM has two processes with one private action each, **a** and **b**, respectively. Every time **pr1** executes **a** it must wait until **pr2** has executed **b** before repeating its action **a**, and vice versa, as the system is symmetric. This informal functional specification can be represented by the state graph shown in Figure 2.4. The state graph contains a special shared action $\tau$, where the synchronization of the two processes happens.

This basic form of synchronization can be modeled by the Petri net in Figure 2.5, which can be implemented with a C-element in hardware [54]. This involves an action, the synchronization transition named $\tau$, which belongs to both processes. When one process is ready to execute $\tau$, it must wait for the other to reach the same point before both execute the action together.

The output of a C-element reflects its inputs when the states of all inputs match. The output remains in this state until all inputs change to the opposite value. In other words, if all inputs are 0, then the output is 0. When all

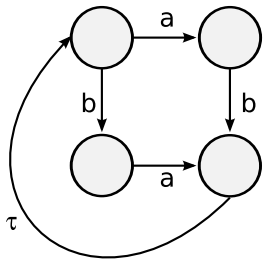



Figure 2.4: Interleaving model for a rendezvous ACM

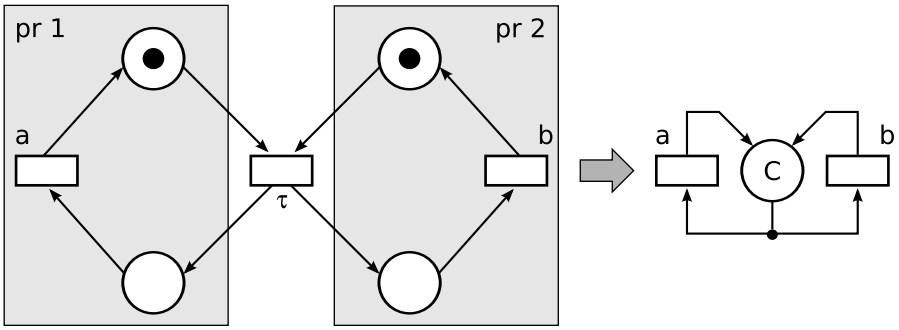



Figure 2.5: Implementing synchronization with a C-element

inputs change to 1, then the output also does. When all inputs change back to 0, the output also does.

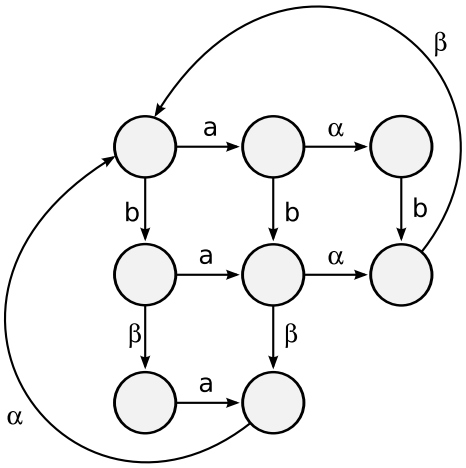
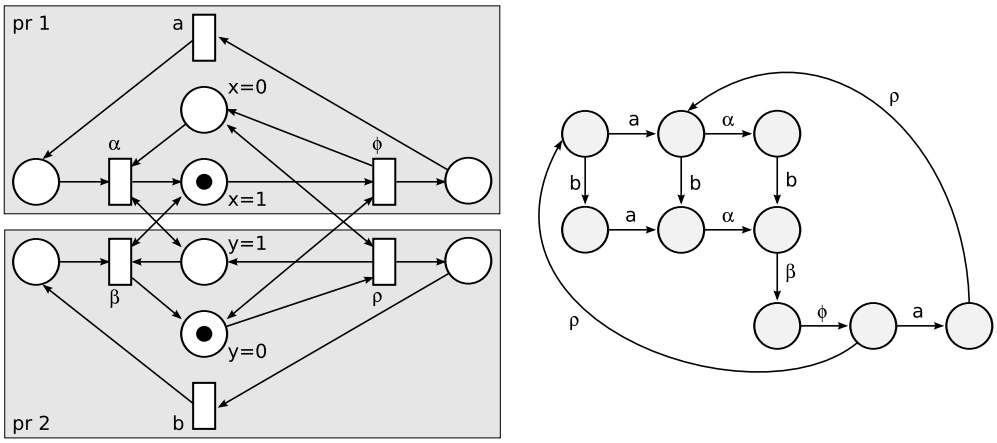



Figure 2.6: Refined model for a rendezvous ACM

Regardless of the functional specification of the ACM, it is possible to

reduce the need for this kind of synchronization to the most basic actions in digital systems [69, 80], i.e. those that can be regarded as atomic. Furthermore, it can be argued that even this type of apparent "hard" synchronization can be implemented without the sharing of an action, atomic or not, by the two processes. For instance, the state graph in Figure 2.4 can be refined into the one in Figure 2.6, where the two processes do not share any action, and the required functional aspect of the synchronization is not lost. Process sharing is removed by replacing $\tau$ with actions $\alpha, \varphi$ for **pr1** and $\beta, \rho$ for **pr2**. This behavior can be implemented by the Petri net model in Figure 2.7.




(a) Petri net model (b) State graph

Figure 2.7: Implementing synchronization with unidirectional control variables

Observe that the state graph generated by this Petri net, shown in Figure 2.7(b), is not isomorphic to the one on Figure 2.6. In order to execute $\alpha$ and $\beta$ concurrently it is necessary to add control actions to serialize the entry and exit of the paths that leads to the execution of $\alpha$ and $\beta$. Since there is no extra actions serializing the entry paths, $\alpha$ always gets enabled before $\beta$, and for this reason, these actions are asymmetric in the state graph of Figure 2.7(b). Observe that $\varphi$ and $\rho$ serialize the exit paths, and they may be executed only after both actions, $\alpha$ and $\beta$, have been executed. However, a projection on events **a**, **b** and $\varphi$ give us a state graph isomorphic to the one on Figure 2.4. Here, the two processes, by sharing places with read or "listening" arcs (i.e. arcs with dual arrows), avoid the sharing of transitions and yet achieve the same functionality of the synchronization point in Figure 2.5. Algorithm 2.2 can be derived [47, 63] for possible software implementation from the Petri net model in Figure 2.7. The two pairs of complementary places can be encoded with variables **x** in **pr1** and **y** in **pr2**.

**Algorithm 2.2** Rendezvous algorithm

```
1: process pr1()
2:   loop
3:     wait until y = 0;
4:     x := 0;
5:     exec a;
6:     wait until y = 1;
7:     x := 1;
8:   end loop
9: end process
```

```
1: process pr2()
2:   loop
3:     wait until x = 0;
4:     y := 1;
5:     exec b;
6:     wait until x = 1;
7:     y := 0;
8:   end loop
9: end process
```

This simple example also illustrates the steps of one of the ACM synthesis method presented in this work. An appropriate interleaving state graph model is derived from a functional specification, and an algorithm-like Petri net model is then derived from the interleaving state graph model.

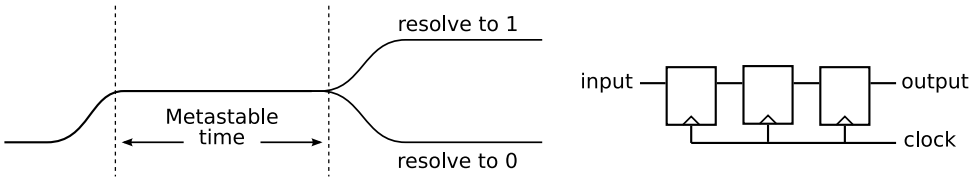
### 2.5.2 Unidirectional control variables

The Petri net model in Figure 2.7(a) effectively specifies "unidirectional control variables", i.e. the value of each of them may be modified (written) by only one of the communicating processes, but can be referenced (read) by both. The binary variables **x** and **y** in Figure 2.7(a), modeled as two pairs of complementary places, are such variables.

Synchronization can be requested by the functional specification or may be required due to an implementation issue. By using unidirectional control variables, it is possible to transfer the synchronization from non-atomic actions, such as the reading and writing of multi-bit data, to actions that can be regarded as atomic or as close to atomic as possible, such as the reading and writing of single-bit control variables. As shown in Figure 2.7(a), all non-atomic actions, represented by transitions **a** and **b**, are fully asynchronous between the two processes. This provides maximum asynchrony for any functional specification. Specifically, if the setting, resetting and referencing of control variables can be regarded as atomic events, the correctness of ACMs becomes easy to prove.

The only possible hazard in a unidirectional control variable of small size (i.e. binary) is associated with metastability [42, 48]. This may happen when a control variable is modified and referenced at about the same time by two asynchronous processes. A metastable binary variable may stay at an analogue value approximately midway between logic 1 and logic 0 for

an indefinite period of time and eventually "resolve" to either 0 or 1 non-deterministically, as shown in Figure 2.8(a).




(a) Sketch of metastability in a variable (b) Typical solution

Figure 2.8: Metastability

In practice, the effects of such metastability, which in modern semiconductor technologies (e.g. CMOS) does not include any oscillatory behavior [62], can be minimized. The typical solution consists of adding a chain of flip-flops to the design in order to reduce the probability of metastability, as shown in Figure 2.8(b). ACM algorithms truthfully implementing appropriate interleaving specifications operate correctly if their control variables are resolved before use. The non-determinism in resolving to either 0 or 1 does not affect the correctness of the ACMs because of the commutative diamonds in the specification. Such techniques, as copying a control variable value through software instructions, drastically reduce the probability of a metastable state persisting until its use. In self-timed solutions, "metastability filters" may be used so that a process may wait until any metastability has been resolved.

In the software world, metastability problems are avoided by test-and-set lock (TSL) instructions which makes the access to a given memory segment atomic. Besides that, the access to memory is controlled by electronic devices that prevent two processes accessing a memory segment at the same time. These devices are called Arbiters. These solutions are usually provided by the hardware and for this reason metastability does not appears on software implementations of unidirectional control variables.

## 2.6 Discussion

Although Xia and Yakovlev were the first to propose a systematic way for the synthesis of multi-cell asynchronous communication mechanisms, they only suggested that it is possible to construct an automatic procedure for their method. In Chapter 3 and 4 we will discuss two automatic procedures for the generation of ACMs. The first approach is based on the generation of state graph specifications that satisfy their interleaving specification requirements. This interleaving specification is introduced in Section 3.1.

The second approach, which is more effective, is based on the generation of the Petri net models without first obtaining the state graph specifications. This second method takes advantage of the very regular structure of the ACMs, and it is based on the definition of *building blocks* that are used to synthesize the Petri net models.

In both cases, the ACMs synthesized in this work are built upon the same assumptions used by Xia and Yakovlev, i.e. any synchronization between the reader and writer processes are restricted to unidirectional control variables.

# Chapter 3

# State space based approach to ACM synthesis

The main motivation of the present work is to introduce a systematic and automatic method for the construction of ACMs having as starting point only a functional specification. The objective is that, in the end of the process, the designer of an asynchronous system will obtain an artifact that provides communication between two asynchronous processes. Such communication scheme will make use of a shared memory, whose access is constrained by a set of binary control variables, to communicate data.

On the attempt achieving that objective, two methods have been obtained. The first one is based on the automatic generation of a graph that represents the state space of the ACM [16, 26]. Such state space is built taking into account a proper interleaving specification and the resulting behavior implements correctly the properties expected from the ACM. The state space is used to synthesize its corresponding Petri net [55, 57] model using an adaptation of the theory of regions [56] for ACMs. The second method is based on the direct generation of the Petri net model without the need of generating its state space. In this chapter, the first method is described.

As stated above, in this method a state graph model is obtained from a functional specification and then its Petri net is synthesized. It differs from the usual way of synthesizing asynchronous circuits, in which a Petri net specification is first obtained, and then its state graph is constructed. This approach is used because the implementation we want to generate, especially in hardware, is something where actions are distributed between components and can be made truly concurrent. Considering this, a Petri net model is closer to the final implementation than its state space. Hence, the generation of the implementation from a Petri net model is more natural.

Figure 3.1 illustrates this technique, which is summarized by the following

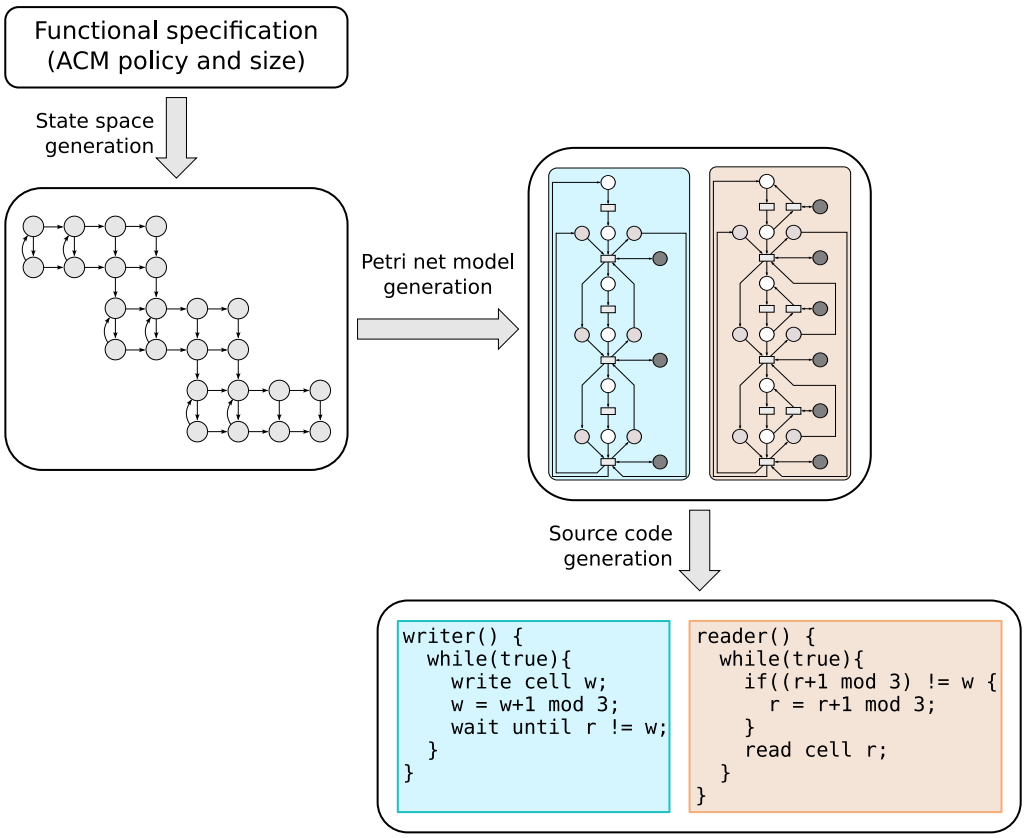



Figure 3.1: Automatic synthesis of ACMs using state space generation

steps:

1. Generate the state graph specification according to a given functional specification;
2. Generate the Petri net model from the state graph using the theory of regions for ACMs;
3. Generate an algorithmic description from the Petri net model;
4. Synthesize software or hardware code implementing that algorithmic description.

In what follows, the generation procedure for the state space of an ACM for the given policy and size as well as the theory of regions for ACMs are detailed. Then the generation of source code from a Petri net model is addressed. Finally, some of the problems with the approach described in this chapter are discussed.

## 3.1 Refined specification with control actions

In order to introduce the type of interleaving specification needed to generate an ACM state graph and the method of obtaining it, a small example will be used. This is the re-reading bounded buffer ACM with three data cells introduced on Section 1.1. The basic interleaving state graph of that ACM is shown in Figure 3.2 and was introduced by Xia and Yakovlev [81]. This state graph includes only the data reading and writing actions.

In Figure 3.2, $rd_i$, $i = 0, 1, 2$, indicates a reading access on the $i^{th}$ cell, and $wr_j$, $j = 0, 1, 2$, indicates a writing access on the $j^{th}$ cell. The states $s_0$ and $s_1$ appear twice in the figure to depict a more readable layout. The entire graph is therefore cyclic. It can be seen that the reader is never forced to wait, and can re-read an item when necessary, while the writer sometimes waits while the reader is accessing a certain cell.

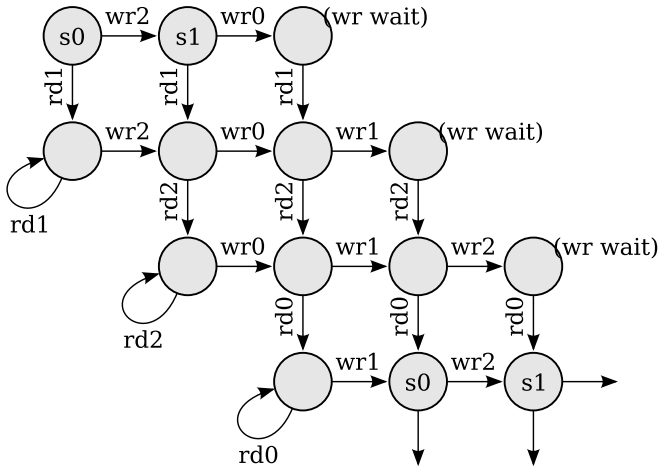



Figure 3.2: Basic state graph specification of an RRBB ACM with 3 cells

This type of state graph is not suitable for synthesizing ACM algorithms [80, 85]. Both the writer and the reader need to make decisions about whether to wait, if the process is required to, or access the ACM. Such decision implies *"hidden actions"*, similar to $\alpha$ and $\beta$ in Figure 2.6, which are not shown in the state graph of Figure 3.2.

Extending the state graph in Figure 3.2 to include the necessary hidden actions produced the refined specification in Figure 3.3. In this figure, $\lambda_{ij}$ indicates the hidden writer action which advances from cell $i$ to cell $j$, and $\mu_{kl}$ indicates the hidden reader action which either advances from cell $k$ to cell $l$ when $k \neq l$ or prepares for the re-reading of cell $k$ in the case of $k = l$. Note that the $\lambda$ actions correspond to lines 5 and 6 of the writer in Algorithm 1.1,

while the $\mu$ actions correspond to lines 4 to 6 of the reader.

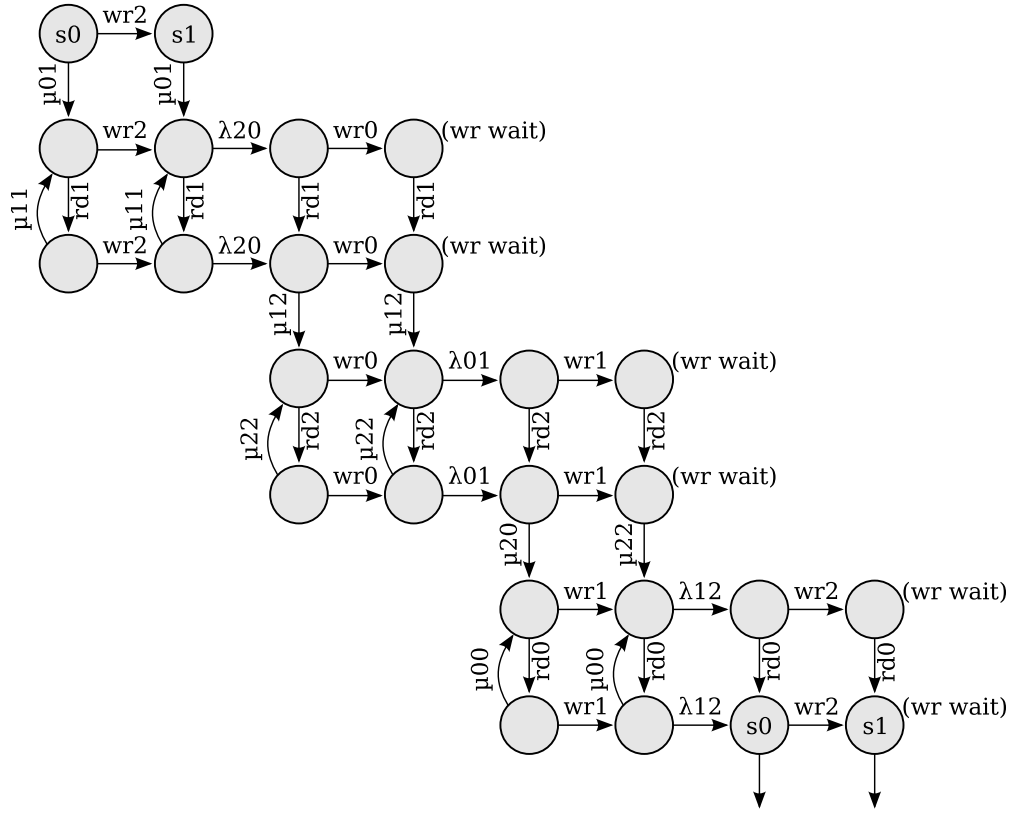



Figure 3.3: State graph including hidden actions of an RRBB ACM with 3 cells

All read and write access actions, denoted by $wr_i$ and $rd_j$, form commutative diamonds with actions of the other partner. This means that these actions are fully concurrent with actions of the other part, maximizing interprocess concurrency. The hidden actions, however, do not have this property. Therefore all the critical synchronization points in the communication are concentrated on them. In order for the resulting ACM to be as concurrent as possible, it is important that the hidden actions take a very small amount of time and be atomic, ideally. It is assumed here that this is accomplished by making such actions the setting and reading of unidirectional control variables of small size.

In [79] it has been shown that when these actions are regarded as non-atomic, not all ACM implementations work according to their specifications. However, some ultra-safe solutions have been found that work correctly even when atomicity is assumed at a lower level, such as the beginning and end of

a control variable set or read operation. As a first effort, we choose to regard control variable actions as atomic.

In Figure 3.3 it is indicated that the silent actions of the writer and reader depend on each other. For instance, whether the next reader silent action is $\mu_{11}$ or $\mu_{12}$ depends on whether $\lambda_{20}$ has completed. And whether $\lambda_{20}$ may start (or the writer must wait) depends on if $\mu_{01}$ has completed. This means that, if using unidirectional control variables, these silent actions set control variables for the other side's silent actions to read.

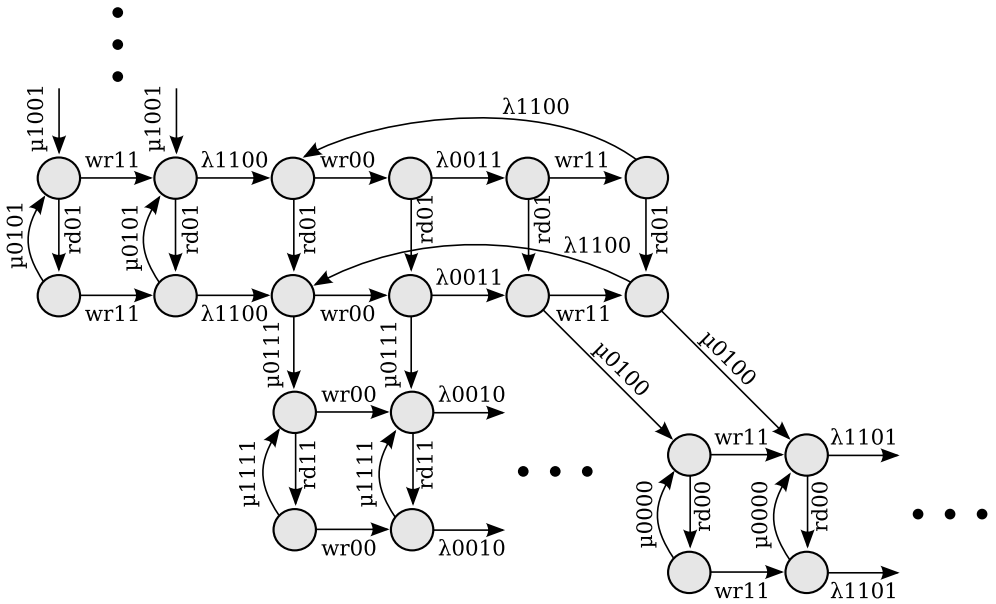



Figure 3.4: Partial state graph of 2x2-cell OWRRBB ACM including hidden actions

Allowing overwriting brings further problems. In the context of ACM, the speed of execution of one process is not related to the other, i.e. no assumption is made about the rates of producing and consuming the data items being transferred. Also, no synchronization between the processes is allowed outside the ACM. Pointing the writer towards the cell to which the reader will next be pointing runs the risk of creating data coherence problems. This problem can be solved by using multiple data slots for the data memory of a single cell. The idea of the multi-slot solution is that each cell is capable to hold more than one data item at the same time and only one of them contains valid data that in the case considered in this work is the slot containing the newest data. If both processes are about to point at the same cell, then they will choose different slots.

The simplest multi-slot cell consists of two data slots. Figure 3.4 shows

part of the state graph, including the silent actions, of a 2-cell OWRRBB ACM using two data slots per cell. The entire graph has 80 states and 160 arcs and is too large to include here. In Definitions 3.4 and 3.5 of Section 3.2 it is explained how this graph is generated.

In Figure 3.4 the notation $\lambda_{ijkl}$ is used to denote the writer moving from the pair [cell,slot] $[i, j]$ to $[k, l]$. And the notation $\mu_{ijkl}$ is used to denote the reader moving from the pair [cell,slot] $[i, j]$ to $[k, l]$. Arcs labeled $\mu_{ijij}$ indicate the reader preparing to re-read when the ACM does not contain any newer item, and the cycles $w_{00} \rightarrow \lambda_{0011} \rightarrow w_{11} \rightarrow \lambda_{1100}$ mean the writer looping without the reader moving. Observe that due to the multi-slot mechanism, both can be pointing at the same cell at a given time, cell 0 in this specific case, however they will be accessing different slots. Depending on where the writer is, the reader may advance to the next cell or continue advancing until it reaches the correct cell. $\mu_{0111}$ and $\mu_{0100}$ branch off to different parts of the state graph. $\mu_{0100}$ appears to be the reader staying in the same cell (cell 0), but it is not, in fact, the case. The reader tries to advance to the next cell (cell 1), finds the writer accessing it, and then advances to the next cell of cell 1, which happens to be cell 0 again in this configuration. In an ACM with more than two cells this action will have a different result.

Obviously, as the size of the ACM increases, or more complex policies are required, it becomes more complicated for a human to draw the state graph describing the behavior of the system.

## 3.2 Deriving the state graph specification

The automatic generation of state graph specifications for ACMs is a fundamental step in the direction towards the synthesis of ACM implementations when the design flow illustrated on Figure 3.1 is used. Such a state graph must satisfy the interleaving properties presented in Section 3.1. It is generated from a functional specification, i.e. it is necessary to provide the type (BB, RRBB, OWBB or OWRRBB) of the ACM and the maximum amount of data items it can hold at the same time. The designer does not need to manage the complexity of specifying how blocking of data access, read or write operation, is avoided and how data coherence is guaranteed during the execution of the system.

This section defines the graph-based model of an ACM corresponding to its functional specification. This specification is defined by two parameters:

1. The re-reading/overwriting policy (e.g. RRBB or OWRRBB);
2. The number of data cells of the ACM.

When designing a synthesis procedure for ACMs state graph models it is important to determine how to calculate the set of possible states of the ACM and how to calculate the successors of a given state. Depending on the function implemented by the ACM, a different number of variables is necessary to distinguish between the states of the generated graph. For instance, in the RRBB ACM there is no need to know the slot number that a process will access since there is only one slot per cell. On the other hand, in ACMs that allow overwriting it is necessary to have this information.

First it will be detailed how to derive the state graph for the RRBB type. A similar same scheme is used in the OWBB and OWRRBB policies, the only difference is the amount of variables needed. The schema for the overwriting policies will be presented afterwards. In Definition 3.1 the notion of an ACM state graph is introduced.

**Definition 3.1 (ACM State Graph Specification)** *A state graph specification for an ACM is a transition system* $[S, T, s_0]$ *such that:*

1. $S$ *is the set of states;*
2. $T \subseteq (S \times S)$ *is the transition relation. We will use* $s \longrightarrow s'$ *to denote that* $(s, s') \in T$*;*
3. $s_0$ *is the initial state.*

A state $s_n$ is said to be reachable from $s_0$ if $s_n = s_0$ or there exists a sequence of actions $(s_0, s_1)(s_1, s_2) \cdots (s_{n-1}, s_n)$ such that for all $0 < m \leq n, (s_{m-1}, s_m) \in T$ and $n > 0$.

## Re-reading ACMs

For an ACM that permits only re-reading, each state of the ACM is defined in terms of the values of the variables that control which cell each process will access, or is accessing. Considering that a process may be ready to access or actually accessing data in $i^{th}$ cell, it is necessary to distinguish between these two internal states of the process. So, a state in the RRBB ACM is determined by four variables, two that denote the cells that the processes are pointing at, and two that specify if a process is accessing the buffer or ready to access it. In Definition 3.2 this concept is formally presented.

**Definition 3.2 (RRBB State)** *A state* $s$ *of an RRBB ACM is a vector* $[i, j, w, r]$*, with* $0 \leq [i, j] < n - 1$ *and* $w, r \in 0, 1$*, where:*

1. $i$ *determines the cell number the writer is pointing at;*

2. *$j$ determines the cell number the reader is pointing at;*

3. *$w$ determines if the writer is ready to access ($w = 0$) or is accessing ($w = 1$) the ACM;*

4. *$r$ determines if the reader is ready to access ($r = 0$) or is accessing ($r = 1$) the ACM.*

$s_{ijwr}$ will be used as a label to state $s$ when it is necessary to show the status of the variables of $s$ explicitly. For example, if a state has the label $s_{0101}$ it means that the writer process is ready to access cell number 0, the next action of the writer is to write data in the cell, the reader is accessing data in cell number 1 and will next carry out a hidden action.

The transition relation is determined by a set of rules that determine the behavior of the ACM. Definition 3.3 captures the formal concepts of the transition rules for RRBB ACMs. For simplicity, $a \oplus 1$ is used to denote $(a + 1) \mod n$ , where $n$ is the number of cells in the ACM.

**Definition 3.3 (Transition rules for RRBB ACMs)** *A transition $t$ is a valid RRBB ACM transition if one of the following applies:*

1. *If $s_{ijwr} \longrightarrow s_{i'jw'r}$ then:*

   *(a)* $(w = 0) \Rightarrow (i' = i \wedge w' = 1)$

   *(b)* $(w = 1 \wedge j \neq i \oplus 1) \Rightarrow (i' = i \oplus 1 \wedge w' = 0)$

   *Condition 1a specifies that if the writer process is ready to access ($w = 0$) the memory cell $i$, then the next action will be to write data into the cell. On the other hand, condition 1b says that if the next action of the writer is a hidden action $\lambda$ (i.e. the process is writing data in the buffer, or has finished doing it) and the reader is not pointing at the next cell, then the writer will point at it. Note that the case $(w = 1 \wedge j = i \oplus 1)$ does not define a valid transition. It corresponds to the case in which the writer waits until the reader advances to another cell, i.e. until $(j \neq i \oplus 1)$.*

2. *If $s_{ijwr} \longrightarrow s_{ij'wr'}$ then:*

   *(a)* $(r = 0) \Rightarrow (j' = j \wedge r' = 1)$

   *(b)* $(r = 1 \wedge i \neq j \oplus 1) \Rightarrow (j' = j \oplus 1 \wedge r' = 0)$

   *(c)* $(r = 1 \wedge i = j \oplus 1) \Rightarrow (j' = j \wedge r' = 0)$

*Condition 2a tells that if the reader is ready to perform data access ($r = 0$) on cell $j$, then the next action is to read the cell. Condition 2b tells that if the reader is going to carry out its hidden action $\mu$ and the writer is not pointing at the next memory cell, then the reader will next point at it. And finally, condition 2c says that if the reader is going to update its control variable and the writer is pointing at the next memory cell, then the reader will prepare to re-read the data in cell $j$.*

3. *No other transitions are valid.*

Observe that by not allowing the writer to advance to the next cell when $(w = 1 \wedge j = i \oplus 1)$ and by setting the reader to re-read the same cell it has just accessed, the situation in which both processes access the same cell simultaneously is avoided, thus preserving data coherence. Also, note that once the writer finishes sending new data, it makes the data available for the reader by advancing to the next cell as soon as possible (i.e. when the reader is not pointing at it). Hence, the last written data item will not be immediately available to the reader only if the buffer is full of non-read data. Since the reader always attempts to access the oldest data not yet read, and the writer is just waiting for the reader to advance to next cell to do it also, freshness is also preserved.

In Figure 3.5 the state graph generated for a 3-cell RRBB ACM using the algorithmic method described above is shown. The entire state graph is isomorphic to the one in Figure 3.3. The initial state is labeled with **2001**, meaning the writer is ready to access the data cell 2, and the reader is ready to perform a silent action.

The generation of the successors of the initial state is done by applying the rules 1a generating the state **2011** and 2b generating the state **2100**. The next step is to generate the successors of such states. Since, in the state **2011**, the execution of the writer satisfies neither 1a nor 1b, the writer cannot be executed, and the execution of the reader reaches the state **2110** by applying rule 2b. The application of rules 1a and 2a in the state **2100** leads to the states **2110** and **2101**, respectively. The execution of the steps above for all states generates the entire state graph of the ACM.

For ACMs that allow overwriting it is necessary to have four extra variables to identify the states: two to identify the data slot and two to specify the index (cell,slot) of the last slot in the pair accessed by the writer. The rules are more complicated, but the principles behind them are the same.

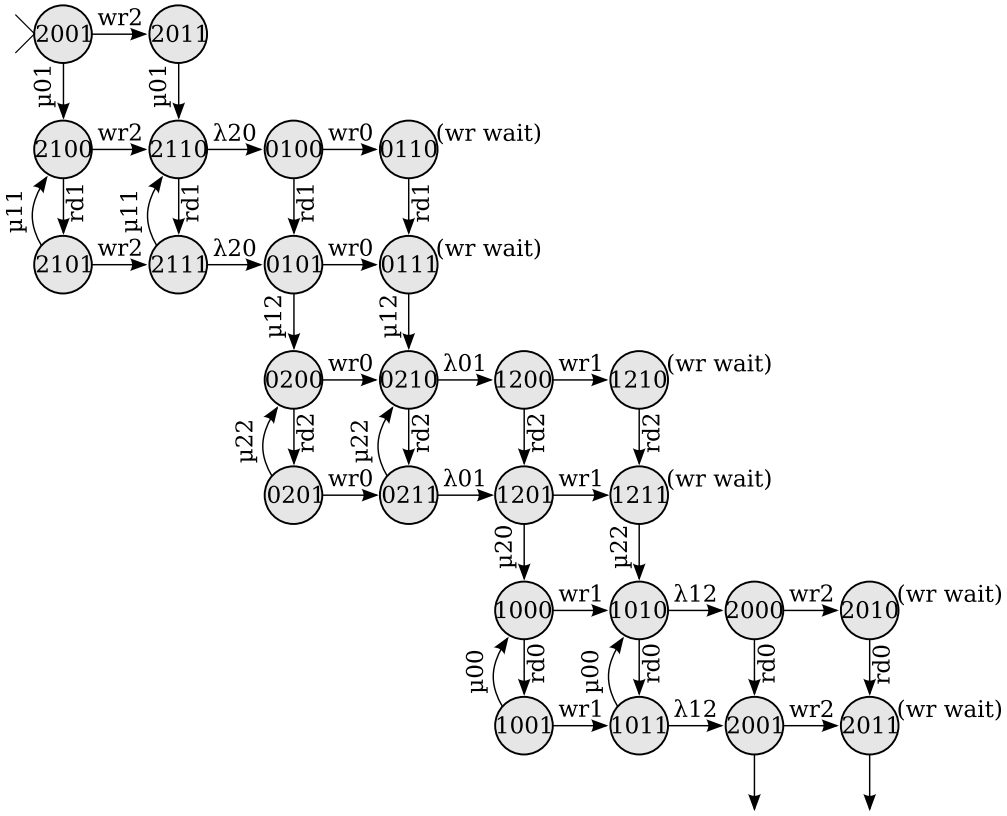



Figure 3.5: Generated state graph of 3-cell RRBB

## Overwriting ACMs

Definitions 3.4 and 3.5 introduce the notion of OWRRBB state and the transition rules for OWRRBB ACMs. These will not be discussed in detail, and are presented here only for completeness. Again, for simplicity, $(Y \bullet y)$ is used to denote that the element $y$ will be appended to the end of a vector $Y$ if $|Y| < n$, where $n$ is the size of the ACM. In the case that $|Y| = n$, $(Y \bullet y)$ indicates that the first element of $Y$ is removed and $y$ is inserted. Also $Y^{\ominus}$ is used to denote the removal of the first element of vector $Y$.

**Definition 3.4 (OWRRBB State)** *A state $s$ of an OWRRBB ACM is a vector $[i, j, w, r, s_i, s_j, o]$, with $0 \leq [i, j, l_i] < n - 1$ and $w, r, s_i, s_j, o \in 0, 1$, where:*

1. *$i$ determines the cell number the writer is pointing at;*

2. *$j$ determines the cell number the reader is pointing at;*

3. *$w$ determines if the writer is ready to access ($w = 0$) or is accessing ($w = 1$) the ACM;*

4. *$r$ determines if the reader is ready to access ($r = 0$) or is accessing ($r = 1$) the ACM;*

5. *$s_i$ determines the slot the writer in pointing at;*

6. *$s_j$ determines the slot the reader in pointing at;*

7. *$o$ determines if the writer is in an overwriting loop.*

To build the state graphs of ACMs that allow overwriting it is necessary to save the history of the last $n$ pairs *(cell,slot)* accessed by the writer that contains non-read data. Here, we define a vector of pairs called $LAST$ to do so.

**Definition 3.5 (Transition rules for OWRRBB ACMs)** *A transition $t$ is a valid OWRRBB ACM transition if one of the following applies:*

1. *If $s_{ijwrs_is_jo} \longrightarrow s_{i'jw'rs'_is_jo'}$ then:*

   (a) $(w = 0) \Rightarrow (i' = i \wedge w' = 1 \wedge s'_i = s_i \wedge o' = o)$

   (b) $(w = 1 \wedge j \neq i \oplus 1) \Rightarrow (i' = i \oplus 1 \wedge w' = 0 \wedge s'_i = 1 \wedge o' = o \wedge LAST = LAST \bullet [i, s_i])$

   (c) $(w = 1 \wedge j = i \oplus 1) \Rightarrow (i' = i \oplus 1 \wedge w' = 0 \wedge s'_i = \neg s_j \wedge o' = 1 \wedge LAST = LAST \bullet [i, s_i])$

   *Condition 1a captures the writer ready to start writing to cell $i$. The values of the control variables do not change, and also the overwriting bit, i.e. if the writer is in an overwriting loop, it will not leave the loop by starting writing to the ACM. And the same is true in the case when the writer is not in the loop. In condition 1b we have the writer advancing to the next cell, by default to slot 1, in a situation in which the reader is not pointing at such a cell. Note that in this case, the condition of being in an overwriting loop does not change either. Also observe that a new pair [cell,slot] is added to the $LAST$ vector. Finally, condition 1c captures the writer entering an overwriting loop. The reader is pointing at the next cell, which indicates that the ACM is full of data, then the writer advances to such a cell, but to the opposite slot to the one the reader is pointing.*

2. *If $s_{ijwrs_is_jo} \longrightarrow s_{ij'wr's_is'_jo'}$ then:*

*(a)* $(r = 0) \Rightarrow (j' = j \wedge r' = 1 \wedge s'_j = s_j \wedge o' = o)$

*(b)* $(r = 1 \wedge |LAST| > 0) \Rightarrow (j' = LAST[0][0] \wedge r' = 0 \wedge s'_j = LAST[0][1] \wedge o' = 0 \wedge LAST = LAST^{\ominus})$

*(c)* $(r = 1 \wedge |LAST| = 0) \Rightarrow (j' = j \wedge r' = 0 \wedge s'_j = s_j \wedge o' = 0)$

*Condition 2a captures the reader starting accessing the ACM, and the control information is not modified. Condition 2b captures the reader advancing to the cell that contains the oldest non-read data. Condition 2c captures the reader preparing to re-read the data in the current cell. Observe that when the reader advances to a new cell, it indicates that the system is not in an overwriting loop. It also indicates that there is a new empty cell, by removing an element from LAST, in the ACM.*

Conditions 1c and 2c together guarantee the preservation of data coherence by avoiding both processes accessing the same slot at the same time. Also, the writer makes the recently written data available for the reader by advancing to the next cell just after it finishes accessing the buffer. If the buffer is full then it invalidates the oldest non-read data. The reader always advances to the pair of cell and slot with the oldest valid data available. So, freshness is also guaranteed. Observe that for an ACM with $n$ cells we understand freshness as related to the last $n$ non-read data items, and not to the last written data item only.

Using the guidelines above a tool that can generate a state graph specification for RRBB, OWBB and OWRRBB ACMs that satisfies the interleaving specification defined in Section 3.1 was implemented. Note that to generate an OWBB ACM, the rules in Definition 3.5 can be applied. It is only necessary to remove rule 2c.

## 3.3 Petri nets synthesis methodology

The problem of synthesis of asynchronous communication algorithms as a problem of synthesizing a Petri net of a certain class is discussed in this section. This class represents the nets that are built as a composition of process nets communicating via specially designated places, called communication places, according to the requirements of unidirectional control variables outlined in Section 2.5.2. The method for the synthesis of communications in Petri nets is based on a more general procedure of synthesizing Petri nets, which uses the theory of regions [56].

The objective of Petri net synthesis is to obtain a Petri net in which transitions are named by the labels of the arcs in the state graph specification, and whose reachability graph is equivalent to the state graph. Different forms of equivalence, such as isomorphism and bisimilarity, have been studied, e.g., in [17].

Informally, such synthesis relates the global states of the state graph with the local states of the system that can be associated with places in the Petri net. More formally, synthesis is based on the concept of regions in transition systems [56], and regions have one-to-one correspondence to places in the synthesized net.

### 3.3.1 Regions

A *region* is a subset of states in which all arcs labeled with the same event $e$ have exactly the same exit/entry relationship. We say that a subset of states $r$ is *entered* by event $e$ if for every arc labeled with $e$ the source state does not belong to $r$ while the destination state is in $r$.

Similarly, $r$ is exited by $e$ if for every $e$-labeled arc the source state is in $r$ but the destination state is outside. In the remaining cases, $e$ is said to be *non-crossing*, by being either *external* or *internal* event for $r$. Thus to become a region a subset $r$ must satisfy *exactly one* of three cases for every event:

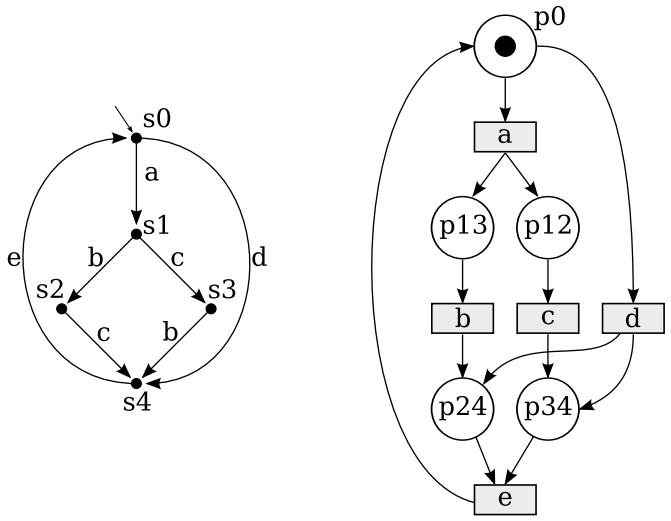
1. enter;
2. exit;
3. non-cross.

In relation to a particular event $e$ a region $r$ is called a pre-region (post-region, co-region) of $e$ if $r$ is exited by (entered by, internal for) $e$.

For example, the set of states $r = \{s_1, s_2\}$ in Figure 3.6(a) is a region, with event $a$ entering $r$, $c$ exiting $r$ and $\{b, d, e\}$ not crossing $r$. However, the set of states $r' = \{s_0, s_1\}$ is not a region, since events $b$ and $c$ have arcs exiting $r'$ and arcs not crossing $r'$.

It is known from [17] that, in order to generate a *safe* Petri net whose reachability graph is isomorphic to a given state graph, the state graph must satisfy the important properties of *state* and *state-event separation*.

Informally, the state separation property requires that for any two different states there exists a region which contains one of the states and does not contain the other. The state-event separation property requires that, for every state $s$ and every event $e$, if the sets of pre-regions and co-regions of $e$




(a) State graph (b) Petri net

Figure 3.6: State graph and generated Petri net.

are included in the set of regions such that each of them contains $s$, then $e$ must be enabled in $s$, i.e. there must be an arc leading from $s$ labeled with $e$.

The basic procedure to produce a *safe* Petri net from a state graph satisfying the above properties is as follows:

1. For each event $e$ in the state graph, a transition named $e$ is created in the Petri net;

2. For each region $r$, a place named $r$ is generated;

3. Place $r$ is connected with a transition $e$ by an arc going from the place (transition) to the transition (place) if region $r$ is pre-region (post-region) for $e$. Place $r$ is connected to $e$ by a bi-directional arc (self-loop) if region $r$ is a co-region for $e$;

4. Place $r$ contains a token in the initial marking iff the corresponding region $r$ contains the initial state of the state graph.

This procedure, if applied, would generate the so-called saturated net [17], since all regions are mapped into corresponding places. A saturated net may have a lot of redundancy, in the sense that some of its places may be removed without disturbing the isomorphism between original state graph and the reachability graph of the synthesized net. Different criteria can be applied

when building a minimal Petri net (in terms of the net size). For example, the criterion to guarantee the state and state-event separation properties, and use only the minimum number of regions is implemented in the Petrify tool [17]. The resulting Petri net reflects the notion of concurrent operation between events forming commutative diamonds in the interleaving (i.e. state graph) form.

Figure 3.6(b) depicts a Petri net obtained from the synthesis of the state graph in Figure 3.6(a). The sub-indices of the places denote the states included in the region represented by the place (e.g. $p_{13} = \{s_1, s_3\}$). Note that not all regions are used for synthesis. The set of states $p_{1234} = \{s_1, s_2, s_3, s_4\}$ is also a region, but it would be redundant if added to the Petri net.

### 3.3.2 Synthesis of ACMs

Nets obtained by the aforementioned synthesis method do not necessarily satisfy the intuitive requirement of the system composed of processes interacting via unidirectional variables, or in other words interacting by reading some of each other's local states.

For the example of the RRBB ACM, the goal is to obtain a Petri net that consists of two sub-nets, one containing events that are labeled with $wr$ and $\lambda_{ij}$ and the other with events labeled with $rd$ and $\mu_{kl}$.

The problem for ACM synthesis can be stated as follows: given a state graph in which each event is associated to a process, derive a Petri net that is the composition of a set of subnets, each one representing a process, in such a way that the marking of each place is only modified by one of the processes.

This formulation requires the association of each place to a process. Other processes can only interact with the place via read arcs.

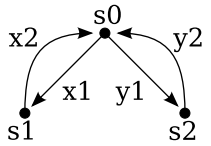
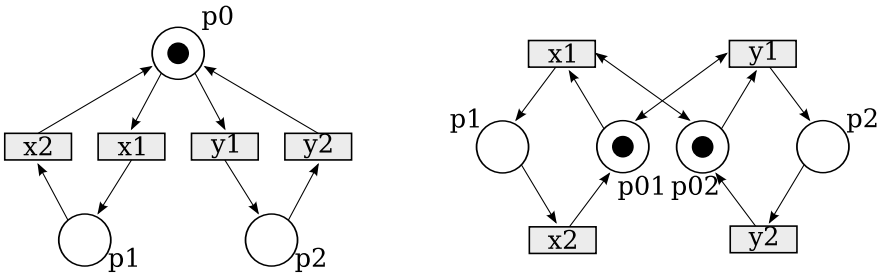



Figure 3.7: State graph for ACM regions

The constraints imposed by ACMs can be illustrated by the example depicted in Figure 3.7 and 3.8. Figure 3.8(a) shows the Petri net obtained

from the synthesis of the state graph in Figure 3.7. The state graph has events from two processes, $x$ (events $x1$ and $x2$) and $y$ (events $y1$ and $y2$). Note that each place corresponds to one state.




(a) Petri net (b) Petri net with ACM-regions

Figure 3.8: State graph and generated Petri net with ACM-regions

Unfortunately, the net in Figure 3.8(a) does not fulfill the requirements for an ACM, since place $p_0$ cannot be associated to any process (it can be modified by both processes $x$ and $y$). By using a different set of regions, the net in Figure 3.8(b) can be obtained. In this net, places $p_1$ and $p_{02}$ belong to process $x$, whereas places $p_2$ and $p_{01}$ belong to process $y$. Note that the communication is produced via the read arcs $p_{01} \leftrightarrow x1$ and $p_{02} \leftrightarrow y1$.

Therefore, the synthesis of ACMs can be performed by using a restricted version of regions called *ACM-regions*. A set of states $r$ is an ACM-region if:

1. $r$ is a region, and;
2. if $e_1$ and $e_2$ are two events that *cross* $r$, then $e_1$ and $e_2$ must belong to the same process.

Thus, the synthesis method for ACMs can be implemented as a slight variation of the method presented in [17], using ACM-regions instead of regions. Petrify was modified to include this technique.

## 3.4 Case studies

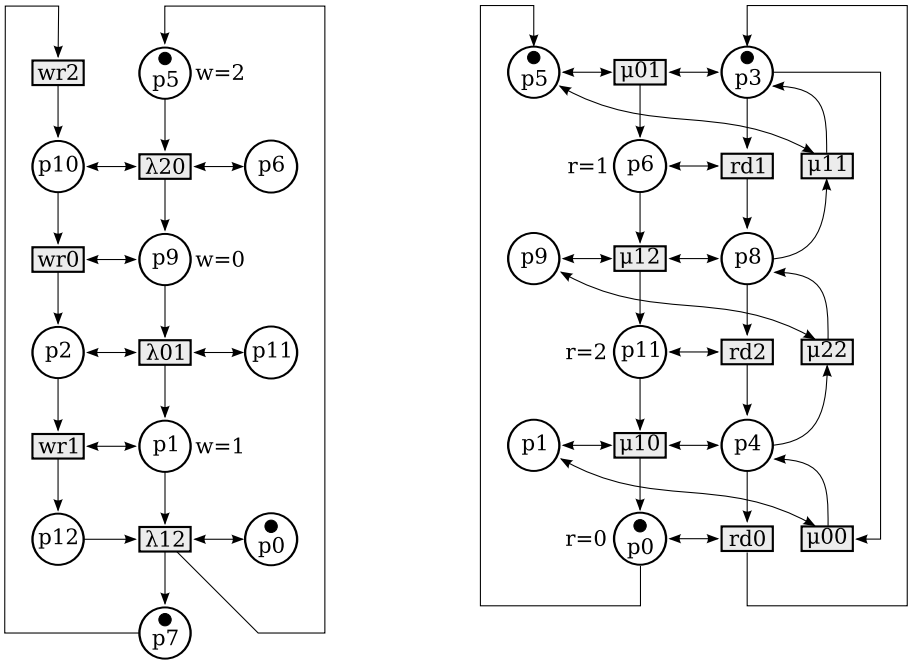
This section shows two examples of application of the methodology described in this chapter, from the initial specification to the generation of the Petri net modeling the behavior of each process. Additionally, we will also give some hints on how to derive an algorithmic description from the Petri net model. This last step will be detailed later.

The designer of the system is responsible for providing a functional specification of the ACM. In the tool we developed this is done by passing as parameters a string specifying the ACM policy, i.e. RRBB, OWBB or OWRRBB, and an integer specifying its number of cells.

With these two pieces of information the state graph specification of the desired ACM is automatically generated. The output of the tool is a textual description which can be used as input to *Petrify*, which implements the technique described in Section 3.3. *Petrify* then generates a Petri net model that represents two concurrent processes that communicate through unidirectional variables. Finally the net is transformed into an algorithmic description that can be easily mapped into some hardware description language like Verilog, VHDL or SystemC or into some software language like C, C++ or Java.

### 3.4.1 3-cell RRBB

The first example is the 3-cell RRBB introduced previously. The generated state graph given in Figure 3.3, and the Petri net obtained from it is presented in Figure 3.9.




(a) Writer process (b) Reader process

Figure 3.9: Petri net model for a 3-cell RRBB

Figures 3.3 and 3.9 indicate that the silent actions of the writer and reader depend on each other. For instance, whether the next reader silent action will be $\mu 11$ or $\mu 12$ depends on whether $\lambda 20$ has completed, and whether $\lambda 20$ may start (or the writer must wait) depends on whether $\mu 11$ has completed. This means that these silent actions set unidirectional control variables for the other side silent actions to read. The simplest set of unidirectional control variables included in the $\mu$ and $\lambda$ actions consists of one writer index variable $w$ and one reader index variable $r$.

Figures 3.9(a) and 3.9(b) show the Petri nets for the writer and reader processes, respectively. The places with the same label are the same, and they are presented separately only because of aesthetic reasons. A token in p5 means $w = 2$, and a token in p6 means $r = 1$. Those Petri nets where obtained by Petrify using the ACM-regions procedure described in Section 3.3.

In the initial state the writer is ready to access cell 2, with transition $wr2$ enabled, and the reader is ready to update the value of its control variable from 0 to 1.

The writer presents a sequential behavior, i.e. it enters into a loop writing data into the cells (alternating this with a silent action) 2, 0 and 1 successively. If the writer is pointing at $i^{th}$ cell and the reader is pointing at $(i \oplus 1)^{th}$ cell, then the writer will wait until it can perform the silent action without running the risk of compromising data coherence.

The reader will also enter a loop and read data from cells 1, 2 and 0. But instead of waiting to preserve data coherence, it will execute one of the transitions $\mu 00$, $\mu 11$ or $\mu 22$, depending on the cell it read before, thus preparing to re-read.

Both choices, between two possible $\mu$ actions and between a $\lambda$ action and writer waiting, are determined by how the current values of $w$ and $r$ compare with each other. For instance, if the reader is pointing at the next cell of the writer, i.e. $r = w \oplus 1$ , then the writer must wait until the reader has changed $r$ before proceeding. Note that here we are assuming the ACM is initialized with some data.

By mapping each place into a Boolean variable, an algorithmic description can be obtained from the Petri net model. For example, transition $\lambda 20$ on the writer process is enabled if there is a token in places p10, p5 and p6. If these places are interpreted as Boolean variables, then $\lambda 20$ can be mapped into a piece of code like:

```
1: if p10 ∧ p5 ∧ p6 then
2:     p5 := false;
3:     p9 := true;
```

4: **end if**

The **if** statement represents the pre-conditions of the transition, i.e. the condition is true if there is a token in places p10, p5 and p6. The statements inside the condition represent the transition removing a token from place p5 (p5 := false) and putting a new token in place p9 (p9 := true). With this approach, an algorithmic description for a hardware or software implementation for each process can be easily obtained.

The first step consists of defining the input, output and internal signals of each process. The input signals of a process are the control variables of its partner that needs to be read by the process, i.e. its test places. For instance, the input signals for the writer and reader are given by the set of places {p0, p6, p11} and {p1, p5, p9}, respectively. The output signals of a process are the control variables that need to be tested by its partner. So, the output of the writer is the input of the reader and vice-versa. Finally, the internal signals are represented by the rest of the places of the process. The initialization of the signals is done according to the initial marking of the place that models the signal.

Each process consists of a single threaded loop in the form of a **case** construct. Each condition in the **case** is given by the pre-conditions of one transition of the Petri net model as explained above. If the transition represented by the **case** statement models an access action, an additional statement is added to the **case** body to perform the data access. Algorithm 3.1 introduces a general schema that can be used to obtain a possible implementation for both processes of the ACM. Each condition is given in terms of the pre-set of a certain transition of the process model and its body reflects the modifications on the markings of the places in the pre-set and post-set of the transition, just as explained before. In other words, it mimics the Petri net model behavior. By repeating the general structure for all transitions, an algorithmic description of the RRBB ACM with 3 cells is obtained.

In an implementation as in Algorithm 3.1, the access to the control variables does not need to be protected. Due to the fact that the variables are binary, metastability problems are resolved in few cycles and the value obtained from reading such variable will be valid, even in the presence of metastability.

It is also possible to obtain an algorithmic description that is more adequate to implement as a software artifact. The sum of the tokens on places p1, p5 and p9 (the writer's output signals) is always equal to 1 for any reachable marking of the Petri net. More than this, p1 is marked before p5 and after p9 are marked. p5 is marked before p9 and after p1. And p9 is marked before p5 and after p1. We can interpret these places as a 3-value control

**Algorithm 3.1** Algorithmic descriptions for the *writer* and *reader* processes

```
1:  module write()
2:    input_w : {p | • p = p•};
3:    output_w : {p | p ∈ input_r};
4:    internal : {p | p ∉ input_w ∧
      p ∉ output_w};
5:    case
6:      ⋀^p_{p∈•wr2}: /* wr2 */
7:        write on cell 2;
8:        ∀p ∈ •t_0 p := false;
9:        ∀p ∈ t_0• p := true;
10:     ⋀^p_{p∈•λ20}: /* λ20 */
11:       ···
12:   end case
13: end module
```

```
1:  module read()
2:    input_r : {p | • p = p•};
3:    output_r : {p | p ∈ input_w};
4:    internal : {p | p ∉ input_r∧p ∉
      output_r};
5:    case
6:      ⋀^p_{p∈•rd0}: /* rd0 */
7:        read from cell 0;
8:        ∀p ∈ •t_0 p := false;
9:        ∀p ∈ t_0• p := true;
10:     ⋀^p_{p∈•μ01}: /* μ01 */
11:       ···
12:   end case
13: end module
```

variable. Also, we can infer some sort of sequence between the values that are assigned to the control variable. The same reasoning is valid for places p0, p6 and p11.

Following this line of reasoning, we can map the transitions wr2 and $\lambda 20$ into the following sequence of code:

```
1: write on cell 2;
2: wait until r = 1;
3: w := 0;
```

By applying the same idea to the other transitions, it is possible to obtain the code that implements the access to each cell in the ACM. A description following this style is clearer than the previous one, however it is not optimized for the implementation of software artifacts. Observe that it presents some redundancy that can be removed by incrementing the value of the control variables instead of using constant values. So, Algorithm 3.2 can be obtained. This is the same as in Algorithm 1.1, and it is reproduced here only to facilitate a comparison with the previous version.

Note that Algorithm 3.2 is independent of the size of the ACM. It is only necessary to set the value of $n$ correctly during the initialization of the system. Also, $w$ and $r$ should be initialized properly, say, with $n-1$ and $n-2$, respectively.

However, such implementation is appropriate only for software artifacts. In the hardware domain, it is more difficult to implement and protect $n$-valued variables, and algorithms in the form of Algorithm 3.1 are more ade-

**Algorithm 3.2** 3-cells RRBB ACM, software implementation

```
1: module write()
2:    wait until (w + 1) mod n ≠ r;
3:    w := (w + 1) mod n;
4:    write on cell w;
5: end module
```

```
1: module read()
2:    if (r + 1) mod n ≠ w then
3:       r := (r + 1) mod n;
4:    end if
5:    read from cell r;
6: end module
```

quate, although more difficult to obtain.

### 3.4.2 4-cell OWBB

The advantage of our method is clearer when trying to specify an ACM of larger size and a more complex function type. For example, specifying manually a state graph of a 4-cell OWBB ACM, in which only overwriting is allowed and both processes always access the memory cell containing the oldest item in the ACM, is difficult due to its size.

Figure 3.10 presents a part of the generated state graph for a 4-cell OWBB ACM, with the initial state in node 0. The entire graph has 1120 states and 2208 arcs, and it is too big to be shown here.

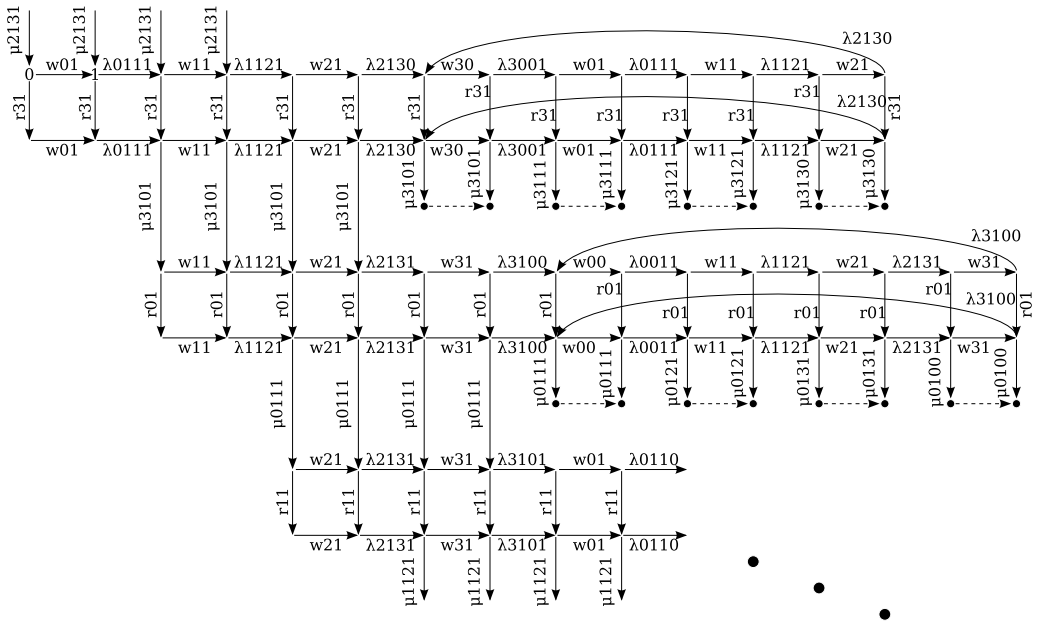



Figure 3.10: Fragment of an OWBB ACM with 4 cells state graph

Again, the resulting state graph can be used to synthesize two Petri nets that specify the behavior of the reader and writer processes. Unfortunately, these are also too large to be included here.

In Table 3.1 it is possible to see how the number of states and arcs of the RRBB, OWBB and OWRRBB ACMs grows with respect to the size of the ACM. Observe that for the OWBB and OWRRBB the number of states is equal. In fact, both ACMs have exactly the same set of reachable states, and differ only in the number of arcs. It is very easy to note that even for very small ACMs it is impractical to produce state graphs by hand.

| ACM size | RRBB (#states / #arcs) | OWBB (#states / #arcs) | OWRRBB (#states / #arcs) |
|---|---|---|---|
| 2 | 12 / 20 | 80 / 144 | 80 / 160 |
| 3 | 24 / 42 | 360 / 696 | 360 / 720 |
| 4 | 48 / 88 | 1120 / 2208 | 1120 / 2240 |
| 5 | 80 / 150 | 3000 / 5960 | 3000 / 6000 |
| 6 | 120 / 228 | 7440 / 14832 | 7440 / 14880 |
| 7 | 168 / 322 | 17640 / 35224 | 17640 / 35280 |
| 8 | 224 / 432 | 40640 / 81216 | 40640 / 81280 |
| 9 | 288 / 558 | 91800 / 183528 | 91800 / 183600 |

Table 3.1: Number of states for RRBB and OWRRBB ACMs

Observe that the solutions generated by Petrify can be used to design parametrizable algorithms. However, at the moment we are not able to do this automatically, and human intervention is required to perform this task. It is our intention to study the generation of closed-form solutions in the future.

## 3.5 Conclusions

In this chapter the generation ACMs through a state graph specification approach is detailed. Firstly the derivation of state graph specification given a functional specification is detailed. Then, the generation of an ACM Petri net model in the form of independent state machines using unidirectional shared variables is described. This last contribution is not a direct contribution of

this thesis. However they are detailed here for the sake of completeness of the synthesis method.

Previously, the steps above were performed manually, which is error-prone due to the size of the state graphs involved. We have automated this process. The method presented here has the great advantage of generating correct by construction Petri net models. On the other hand, for overwriting ACMs with many cells it is not possible to obtain the Petri net model using the presented technique, as the state graphs become too large. In the next chapter an alternative approach avoiding this problem will be discussed.

# Chapter 4

# The modular approach to ACMs

In previous work [26, 16, 81, 85], 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 interleaving 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.

In this chapter the generation of the Petri net model using a modular approach that does not require the explicit enumeration of the state space is introduced. 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. Figure 4.1 shows the design flow for the automatic generation of ACMs.

Compared to the approach introduced in Chapter 3, the modular approach has the advantage of not dealing with the entire state space of the ACM in order to generate the Petri net model. It is obtained in linear time. On the other hand, it requires verifying 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.

## 4.1 Models for verification and implementation

The two basic paradigms on the approach presented in this chapter are *automation* and *correctness*. For that reason, from the functional specification

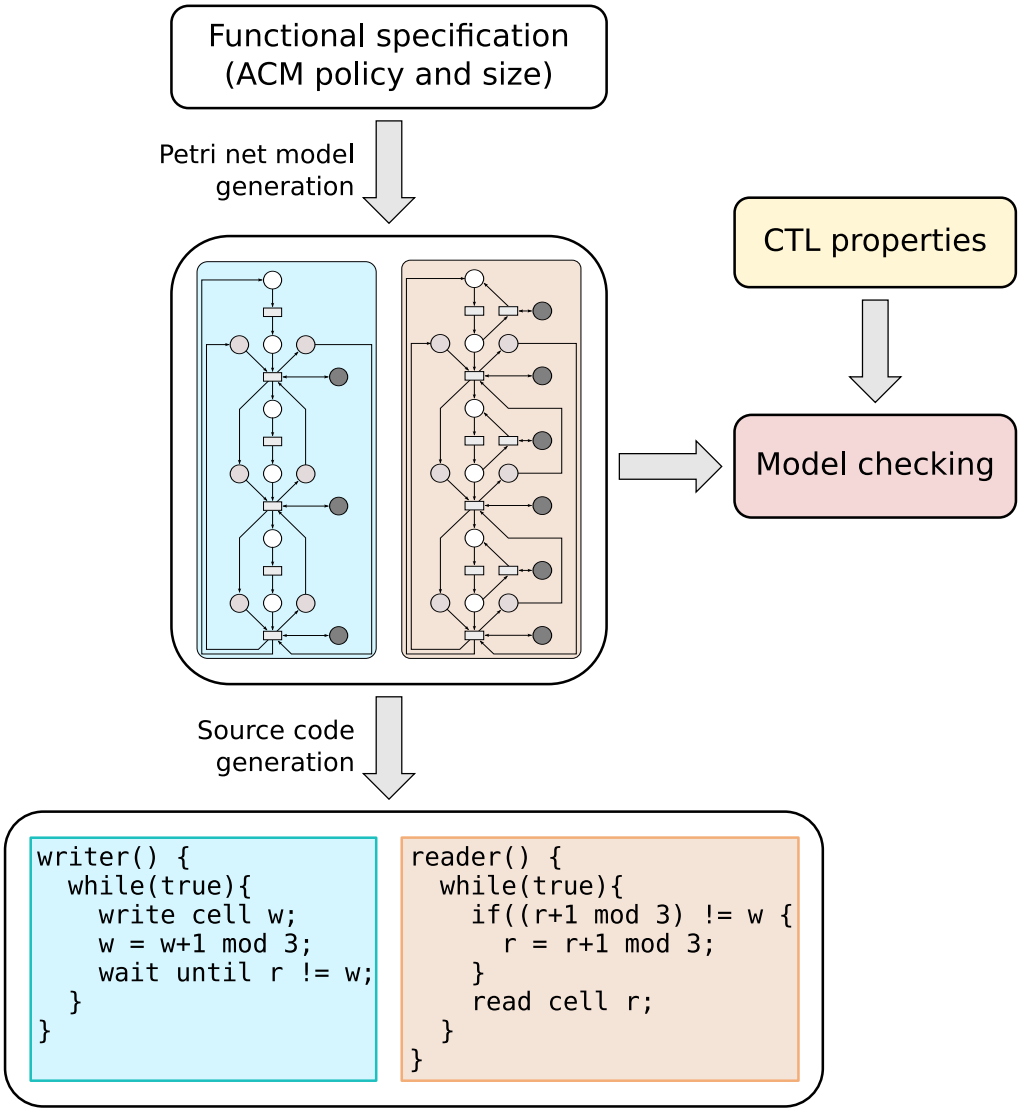



Figure 4.1: The design flow

of an ACM, two formal models can be 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 [49, 50] 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. For simplicity, we will refer to the Cadence SMV as just SMV.

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 sections, the abstract and implementation models for the class of re-reading and overwriting ACMs are presented.

## 4.2 Abstract models

The abstract model for an ACM is specified as a transition system. The state of the ACM is defined by the data items available for reading. For each state, $\sigma$ defines the queue of data stored in the ACM. More specifically, $\sigma$ 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.

$\sigma$ 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 $\sigma$ 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 $\langle \sigma \rangle$.

### 4.2.1 Re-reading ACMs

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 a 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 the data in the ACM has been read before, then it re-reads the last retrieved data item.

Definition 4.1 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 4.1 (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 4.1 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, which can be reused. Specifically, each process consists of a `case` statement in which each condition corresponds to a rule in Definition 4.1. The SMV model obtained from Definition 4.1 will be used in Section 5.3 to verify a lower level specification of the ACM. Next, the specification of the coherence and freshness properties is discussed.

### 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 cell of the shared memory.

In the ACM model described by Definition 4.1, the reader always addresses the data stored in the first position of the ACM, represented by $\sigma$. 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:

- it is addressing the first data item, and
- if the writer is also accessing the ACM, then it is not writing into the first cell.

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:

$$\texttt{AG}\ (a^r \in \sigma \rightarrow (a^r = a_0 \wedge (a^w \in \sigma \rightarrow a^w = a_j \wedge j > 0))) \tag{4.1}$$

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 ($(a^w \in \sigma \rightarrow a^w = a_j \wedge j > 0)$).

**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 $\sigma'$ 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 $\sigma$ nor the writer has stored a new item at 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 $\sigma$, but the writer has added a new item at the tail;
3. $\sigma' = a_1 \cdots a_{j-1} a_j$: in this case the reader has removed a data item from the head of $\sigma$.

The above can be specified by the following CTL formula:

$$\texttt{AG}(|\sigma| = x \rightarrow \texttt{AX}((\sigma' = \sigma^+) \vee (|\sigma'| = x - 1 \wedge \sigma' = \sigma^-))) \tag{4.2}$$

where $\sigma^+$ 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 $\sigma^-$ is used to denote $a_1 \cdots a_{j-1} a_j$. Observe that properties 1 and 2 are captured by the same CTL sub-formula, which is given in the left-hand 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 $\sigma$, it is necessary to generate another CTL formula.

### 4.2.2 Overwriting ACMs

In what follows the expected behavior of an overwriting ACM is described as a transition system. The notation used in Section 4.2.1 will also be used here to denote the states and transitions rules of overwriting ACMs.

In overwriting ACMs the writer can add data in the ACM until it is full. In such a 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 and re-reading is also allowed, then it attempts to re-read the last retrieved data item. In this section we consider the more complex case in which both processes are required not to wait when accessing to the ACM.

The above behavior is formally introduced in Definition 4.2. Rules 1-4 model the behavior of the writer. Rules 5-8 model the behavior of the reader.

**Definition 4.2 (OWRRBB transition rules)** *The behavior set of an OWRRBB ACM is defined by the following set of transitions (n is the number of cells in 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. $\langle a\sigma \rangle \xrightarrow{wr_b(b)} \langle \sigma b^w \rangle$ **if** $|a\sigma| = n$
3. $\langle a^r b\sigma \rangle \xrightarrow{wr_b(c)} \langle a^r \sigma c^w \rangle$ **if** $|ab\sigma| = n$
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. $\langle a^r \sigma \rangle \xrightarrow{rd_e(a)} \langle \sigma \rangle$ **if** $|\sigma| > 0 \ \wedge \ \sigma \neq b^w$
7. $\langle a^r \rangle \xrightarrow{rd_e(a)} \langle a \rangle$
8. $\langle a^r b^w \rangle \xrightarrow{rd_e(a)} \langle a b^w \rangle$

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 the 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 4.2 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, which can be reused. Specifically, each process consists of a `case` statement in which each condition corresponds to a rule in Definition 4.2. The SMV model obtained from Definition 4.2 will be used in Section 5.2.2 to verify a lower level specification of the ACM. Next, the specification of the coherence and freshness properties is discussed. In [25] the RRBB policy was modeled in SMV and verified.

### Coherence

Verifying coherence requires showing that there is no reachable state in the system in which both processes are addressing the same cell of the shared memory. According to Definition 4.2, the reader always addresses the data stored in the head of $\sigma$, while the writer always addresses the end of the tail of $\sigma$. Verifying coherence in this model only requires proving 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 into 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:

$$\texttt{AG}\ (a^r \in \sigma \rightarrow (a^r = a_0 \wedge (a^w \in \sigma \rightarrow a^w = a_j | j > 0))) \tag{4.3}$$

### 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 $\sigma'$ 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 $\sigma$;

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

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

The above can be specified by the following CTL formula:

$$\mathbf{AG}(|\sigma| = x \rightarrow \mathbf{AX}((\sigma' = \sigma^+) \vee (|\sigma'| = x - 1 \wedge \sigma' = \sigma^-))) \tag{4.4}$$

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

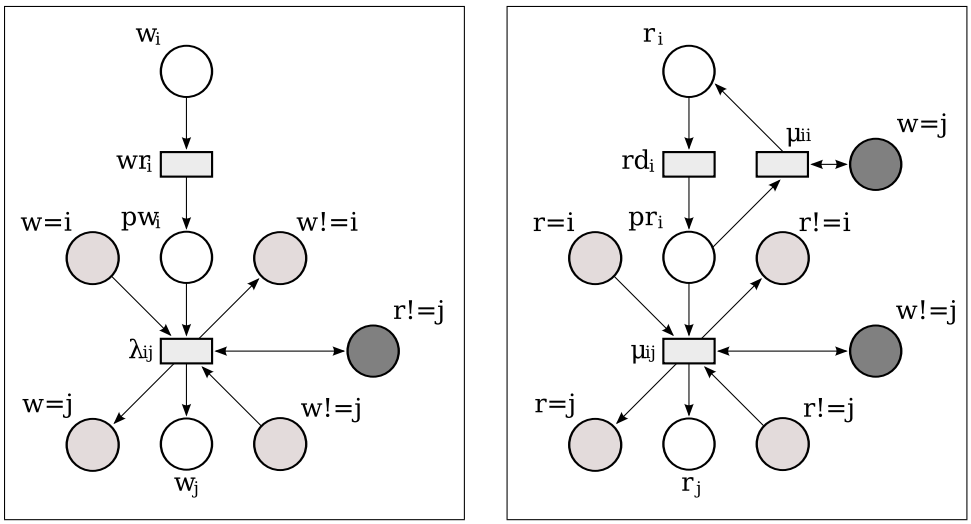
## 4.3 Generating the models for re-reading ACMs

A Petri net model for a 3-cell RRBB ACM will be generated and mapped into a C++ implementation. As stated before, this 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 $i^{th}$ 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 4.2.

In Figure 4.2(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 $i^{th}$ or at $j^{th}$ cell. $\langle r \neq j \rangle$ indicates when the reader is not pointing at $j^{th}$ cell. If transition $\lambda_{ij}$ is enabled, then the reader is not pointing at cell $j$, the writer




(a) writer module (b) reader module

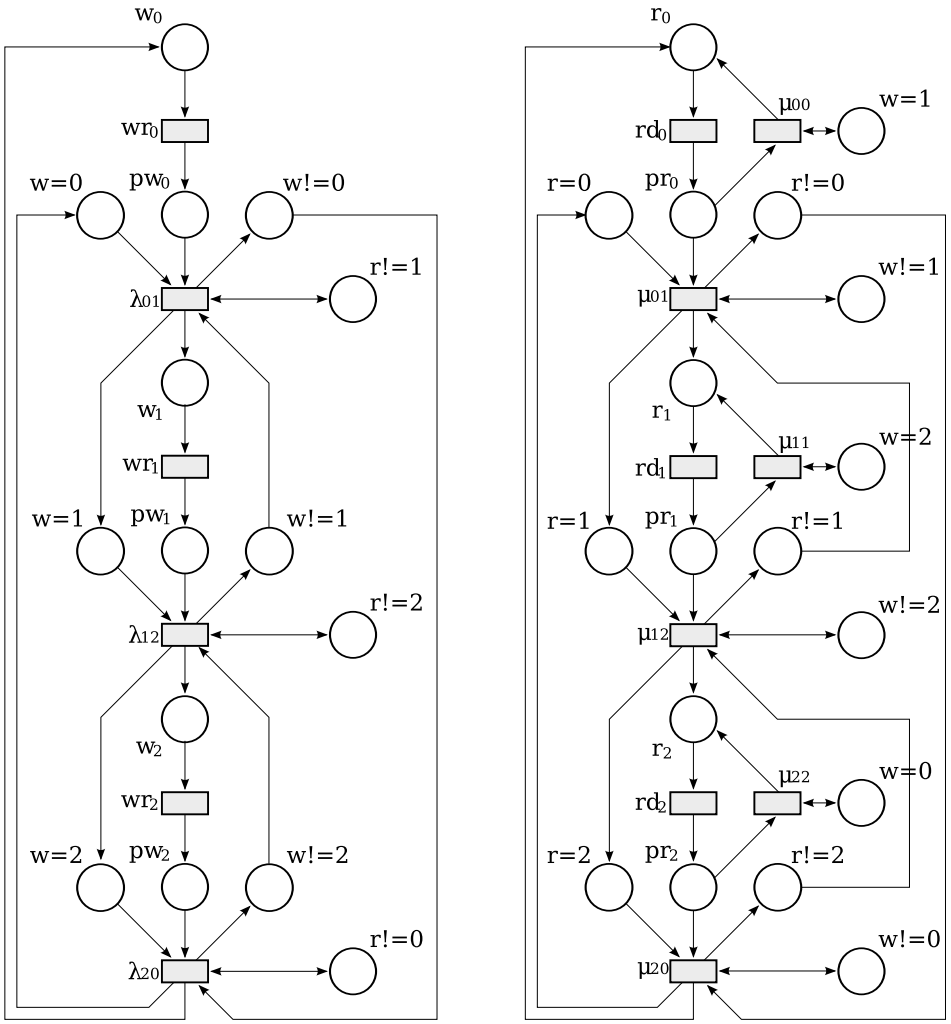
Figure 4.2: Basic modules for the writer and the reader

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 which is shown in Figure 4.2(b). 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, $\mu_{ii}$ and $\mu_{ij}$. Here the decision is based on the current status of the writer, i.e. whether the writer is on the $j^{th}$ cell or not, is 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 on them is always equal to one. Thus only one of the transitions $\mu_{ii}$ and $\mu_{ij}$ can be enabled at a time, i.e. they are not in dynamic conflict.

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 $i$ and $j$ by the correct cell numbers. For example, to instantiate the writer's module to control the access to the $0^{th}$ cell, $i$ is replaced by 0 and $j$ by 1. Connecting the modules requires fusing all the places with the same label. Figure 4.3 depicts the resulting Petri net models for the writer and reader of a 3-cell RRBB ACM.

After creating the processes, they can be connected by also fusing places with same label on both sides. In this case, the shadowed places in each




(a) writer process (b) reader process

Figure 4.3: The write and read processes for a 3-cell RRBB ACM

module will be fused with some place of the other module.

Definition 4.3 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 4.3 (Petri net RRBB module)** *A Petri net module is a tuple*

*$MODULE = (PN, LOC, EXT, T_a, T_c, M_a, M_c)$ 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. *$LOC \bigcap EXT = \emptyset$;*
5. *$T_a \subset T$ is a finite set of transitions such that $t \in T_a \iff t$ models a media access action;*
6. *$T_c \subset T$ is a finite set of transitions such that $t \in T_c \iff t$ models a control action;*
7. *$T_a \bigcup T_c = T$ and $T_a \bigcap T_c = \emptyset$;*
8. *$M_a \subset (T_a \times \mathbb{N})$ is a relation that maps each access transition $t \in T_a$ into an integer that is the number of the cell addressed by $t$;*
9. *$M_c \subset (T_c \times \mathbb{N} \times \mathbb{N})$ is a relation that maps each control transition $t \in T_c$ into a pair of integers that are the numbers of the current and the next cells pointed by the module.*

Definitions 4.4 and 4.5 formally introduce the writer and reader basic modules for RRBB ACMs, respectively.

**Definition 4.4 (RRBB writer module)** *The RRBB writer module is a tuple $WRITER = (PN_w, LOC_w, EXT_w, T_{a_w}, T_{c_w}, M_{a_w}, M_{c_w})$ where:*

1. *$PN_w$ is as defined by Figure 4.2(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. *$T_{a_w} = \{wr_i\}$;*
5. *$T_{c_w} = \{\lambda_{ij}\}$;*

6. $M_{a_w} = \{(wr_i, i)\}$;

7. $M_{c_w} = \{(\lambda_{ij}, i, j)\}$.

**Definition 4.5 (RRBB reader module)** *The RRBB reader module is a tuple* $READER = (PN_r, LOC_r, EXT_r, T_{a_r}, T_{c_r}, M_{a_r}, M_{c_r})$ *where:*

1. $PN_r$ *is as defined by Figure 4.2(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. $T_{a_r} = \{rd_i\}$;

5. $T_{c_r} = \{\mu_{ii}, \mu_{ij}\}$;

6. $M_{a_r} = \{(rd_i, i)\}$;

7. $M_{c_r} = \{(\mu_{ii}, i, i), (\mu_{ij}, i, j)\}$.

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 4.6 captures this.

**Definition 4.6 (Connection for RRBB 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 4.6 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 4.7.

**Definition 4.7 (Initial marking for RRBB ACMs)** *For any Petri net model of an RRBB ACM, its initial marking is defined as follows*

1. $M_0(w_1) = 1$;

2. $M_0(\langle w = 1 \rangle) = 1$;

3. $M_0(\langle w \neq i \rangle) = 1$, *if* $i \neq 1$;

4. $M_0(r_0) = 1$;

5. $M_0(\langle r = 0 \rangle) = 1$;

6. $M_0(\langle r \neq i \rangle) = 1$, *if* $i \neq 0$.

*All the other places are not marked.*

Observe that according to Definition 4.7, the writer is pointing at the $1^{st}$ cell of the ACM and reader is pointing at the $0^{th}$ cell, i.e. the ACM is assumed to be initialized with some data in its $0^{th}$ cell.

## 4.4 Generating the models for overwriting ACMs

Asynchronous communication mechanisms allowing overwriting are considerable more complex than non-overwriting policies, requiring more control variables and a more complex control logic in order to behave properly. As in the case of re-reading ACMs, a set of basic modules which are used to synthesize the writer and reader processes is defined. The creation of each process consists in instantiating a number of these modules, one for each cell, and connecting them. Again, instantiating a module only requires replacing $i$ and $j$ by the correct cell numbers, and connecting the modules requires to fuse all the places with the same label.

The configuration of overwriting ACMs differs from the re-reading ACMs substantially. In RRBB ACMs the shared memory used to provide communication between the processes consists of a simple circular buffer. This was enough to preserve coherence and freshness properties. On OWRRBB and OWBB ACMs, this is not enough. The shared memory also consists of a circular buffer, but in this case each cell in the buffer is composed of two slots. Each cell can hold two data items at a time and only one of them is considered as valid. This is necessary to avoid data coherence problems when both processes are about to access the same cell in the buffer. In this case, the processes will address different slots of the cell.

In an OWRRBB ACM, allowing overwriting and re-reading implies the following behavior:

- The writer first accesses the shared memory and then advances to the next cell. If the reader is not accessing that cell, then the writer will address the slot opposite to the one the writer addressed the last time it accessed the cell. On the other hand, if the reader is accessing that cell, the writer addresses the slot opposite to the slot the reader is addressing. In the latter case, the writer indicates that it engaged in overwriting.

- The reader first advances to the next cell and then performs the data transfer. If the writer is not engaged in overwriting, the reader always addresses the last slot addressed by the writer in the reader's next cell. On the other hand, the reader advances to the writer's next cell and addresses the last slot pointed by the writer, and indicates that the overwriting cycle has finished.

In order to support the behavior above, it is necessary to modify Definition 4.3 to introduce multiple slots support in the Petri net modules. Definition 4.8 formally describes a Petri net module that can be used to build models of overwriting ACMs.

**Definition 4.8 (Petri net OWRRBB module)** *A Petri net module is a tuple* $MODULE = (PN, LOC, EXT, T_a, T_c, M_a, M_c)$ *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. $LOC \bigcap EXT = \emptyset$*;*

5. $T_a \subset T$ *is a finite set of transitions such that* $t \in T_a \iff t$ *models a media access action;*

6. $T_c \subset T$ *is a finite set of transitions such that* $t \in T_c \iff t$ *models a control action;*

7. $T_a \bigcup T_c = T$ *and* $T_a \bigcap T_c = \emptyset$*;*

8. $M_a \subset (T_a \times (\mathbb{N} \times \mathbb{N}))$ *is a relation that maps each access transition* $t \in T_a$ *into a pair of integers that represent the number of the cell and the slot addressed by* $t$*;*

9. $M_c \subset (T_c \times (\mathbb{N} \times \mathbb{N}) \times (\mathbb{N} \times \mathbb{N}))$ *is a relation that maps each control transition* $t \in T_c$ *into two pair of integers. The first models the current cell and slot pair pointed by the module, while the second models the values of the next cell and slot pair to be pointed.*

Observe that Definition 4.8 differs from 4.3 only on items $M_a$ and $M_c$. This is necessary due to the fact that in overwriting ACMs it is not enough to indicate the current and next cells a process is pointing to. It is necessary to indicate the current and next pairs of (cell, slot) the process is pointing to.

In what follows, the modules used to build the writer and reader processes will be defined. Also, the procedure to instantiate and connect the modules is introduced.

### 4.4.1 The writer module

As stated before, in an overwriting ACM the writer first accesses the shared memory, storing a new data item in it, and then it advances to the next cell. Which cell is the next one depends on which cell the reader is addressing. Suppose that the writer is addressing the pair of cell and slot $(i, s)$, with $i = 0..n-1$ and $s = 0, 1$ at a certain time. If the reader is not addressing the $j^{th}$ cell, where $j = i + 1 \mod n$, the writer will address the cell $(j, s')$ at the next time, where $s'$ is the slot opposite to the slot addressed by the writer the last time the $j^{th}$ cell was accessed. On the other hand, if the reader is

addressing the cell $(j, s')$, then the writer should next address the pair $(j, \overline{s'})$. In this case, the writer indicates that it has engaged in overwriting.

Due to the need of extra control variables to correctly indicate which slot in the cell the writer is addressing and if it has engaged into an overwriting cycle, the basic modules used to build a Petri net model of an overwriting ACM writer process is considerable more complex than the basic module for re-reading only ACMs. However, the basic behavior is the same: write a new data item, and then advance to another cell.

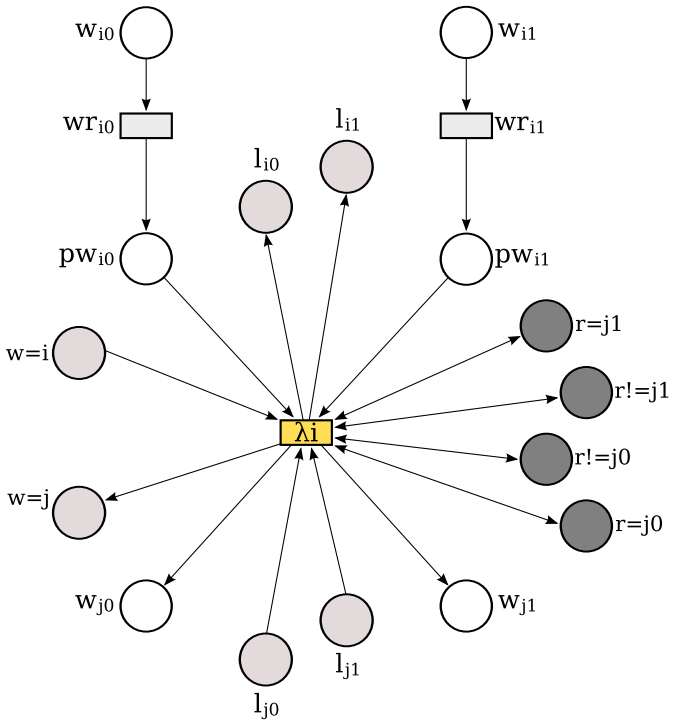
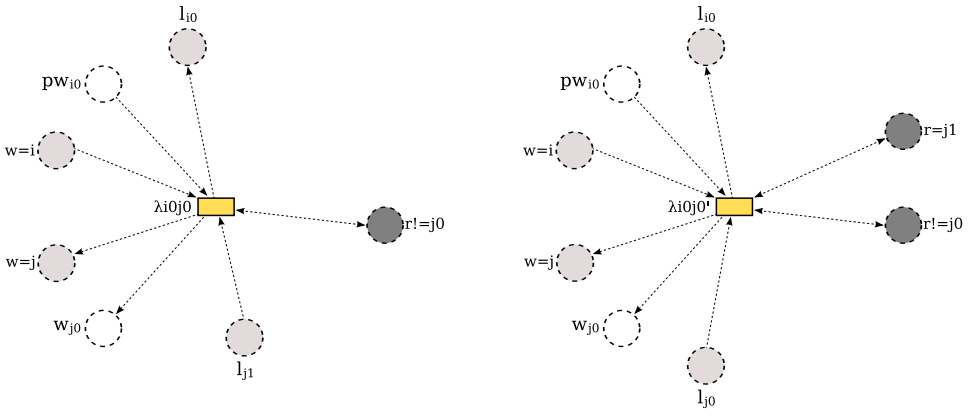



Figure 4.4: The writer module with compressed control actions

In Figure 4.4 this basic behavior is captured as a Petri net. The data access actions are modeled by the transitions $wr_{i0}$ and $wr_{i1}$, which model the action of writing a new data into the pairs of cell and slot $(i, 0)$ and $(i, 1)$, respectively. The control actions are abstracted by the transition $\lambda_i$. The places $w_{i0}$, $w_{i1}$, $pw_{i0}$, $pw_{i1}$, $w_{j0}$ and $W_{j1}$ are internal to the writer process, and they function as a program counter indicating the next action to be performed by the process. The places $w = 1$, $w = j$, $l_{i0}$, $l_{i1}$, $l_{j0}$ and $l_{j0}$ model the control variables of the writer process. While $w = 1$ and $w = j$ indicate which cell the process is addressing at a given time, the others are used to indicate the last slot addressed by the process when it was pointing to $i^{th}$ or

to $j^{th}$ cell. Finally, the places $r = j0$, $r! = j0$, $r = j1$ and $r! = j1$ model the control variables of the reader process, indicating its status.

Note that in Figure 4.4 the transition $\lambda_i$ represents the set of all $\lambda$ transitions in the module. Figures 4.11 – 4.14 detail the possible behavior of each $\lambda$ transition.



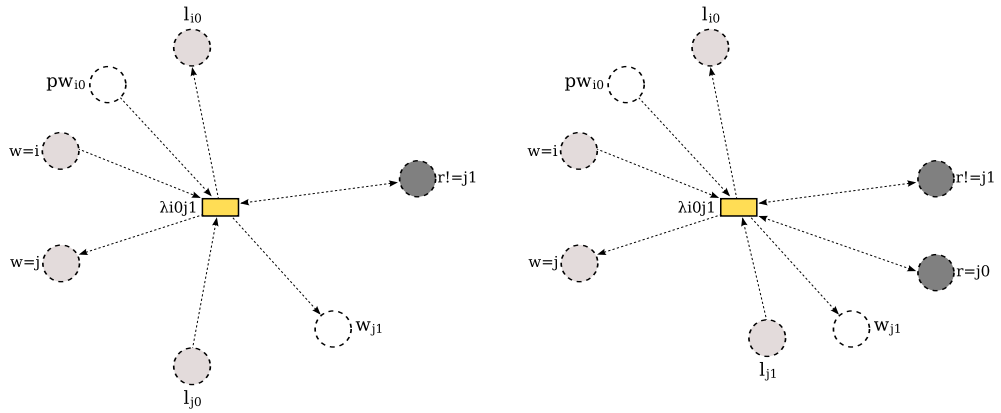

(a) $\lambda_{i0j0}$ (b) $\lambda_{i0j0'}$

Figure 4.5: Control actions for the writer – part 1

In Figure 4.5(a) the transition $\lambda_{i0j0}$ models the writer moving from the pair $(i, 0)$ to the pair $(j, 0)$ when the reader is not addressing it and the last pair addressed in the $j^{th}$ cell was $(j, 1)$. In Figure 4.5(b) $\lambda_{i0j0'}$ does basically the same, but in this case the reader is addressing $(j, 1)$ and the writer has accessed $(j, 0)$ on the previous cycle. Then it is necessary to overwrite the contents of $(j, 0)$ again. Observe that $\lambda_{i0j0'}$ does not indicate the beginning of an overwriting cycle, this is also done by $\lambda_{i0j0}$ in the case of the existence of tokens in $\lambda_{j1}$ and $r = j1$, which is not explicitly tested by $\lambda_{i0j0}$. Also note that the indication of the writer engaging in overwriting is not explicit in the model: this is only indicated to the reader by the existence of a token in place $l_{is}$ when the reader is pointing to $(i, \overline{s})$. This is enough to determine the occurrence of overwriting.

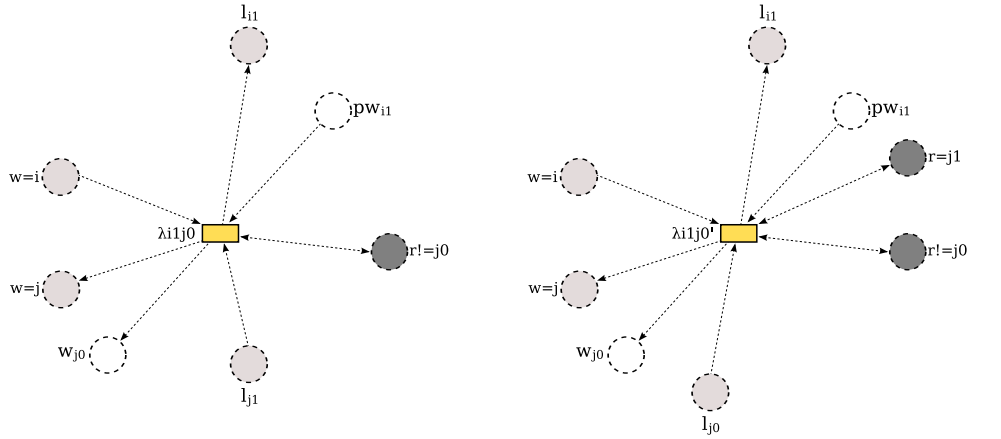
In Figures 4.6(a) and 4.6(b) the transitions $\lambda_{i0j1}$ and $\lambda_{i0j1'}$ are introduced. They both model the writer moving from $(i, 0)$ to $(j, 1)$. As before, $\lambda_{i0j1'}$ is fired when the writer has already engaged in an overwriting cycle and $\lambda_{i0j1}$ is fired when there is no overwriting yet, or when the overwriting cycle is about to start.

Finally, in Figures 4.7 and 4.8 the transitions $\lambda_{i1j0}$, $\lambda_{i1j0'}$, $\lambda_{i1j1}$ and $\lambda_{i1j1'}$ are introduced. They model the writer moving from $(i, 1)$ to $(j, 0)$ and from




(a) $\lambda_{i0j1}$ (b) $\lambda_{i0j1'}$

Figure 4.6: Control actions for the writer – part 2

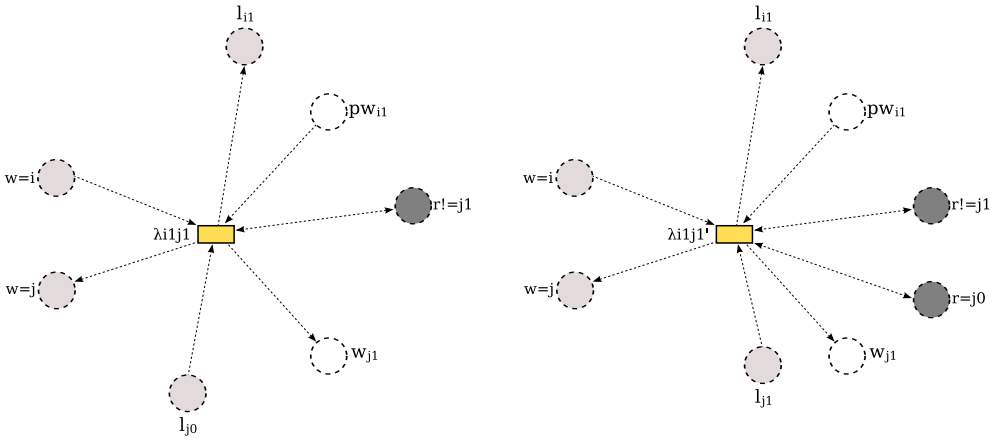



(a) $\lambda_{i1j0}$ (b) $\lambda_{i1j0'}$

Figure 4.7: Control actions for the writer – part 3

$(i, 1)$ to $(j, 1)$. The reasoning used before also applies here. However, in these cases the writer is releasing a data item in $(i, 1)$ instead of $(i, 0)$ as is in the two first cases.

Figures 4.5 – 4.8 together model all the behavior allowed for the writer when it is executing a control action. Replacing the transition $\lambda_i$ in Figure 4.4 by the combination of all of them results is the Petri net module in Figure 4.9. It is not difficult to see that when the size of the ACM grows, it becomes more and more complex to understand the behavior of its Petri net model and, which is more important, to build it manually. Up to now, only the




(a) $\lambda_{i1j1}$ (b) $\lambda_{i1j1'}$

Figure 4.8: Control actions for the writer – part 4

model for the writer has been considered.

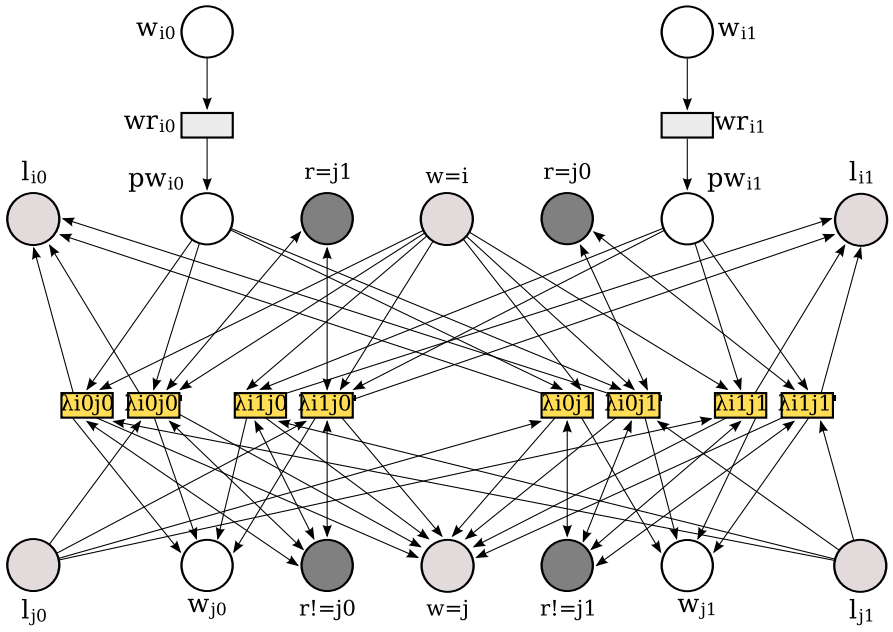



Figure 4.9: The complete writer module

The Petri net module for the writer process of an OWRRBB ACM is formally introduced in Definition 4.9.

**Definition 4.9 (OWRRBB writer module)** *The OWRRBB writer module is a tuple* $WRITER = (PN_w, LOC_w, EXT_w, T_{a_w}, T_{c_w}, M_{a_w}, M_{c_w})$ *where:*

1. $PN_w$ *is as defined by Figure 4.9;*
2. $LOC_w = \{\langle w = i\rangle, \langle w = j\rangle, \langle l_{i0}\rangle, \langle l_{i1}\rangle, \langle l_{j0}\rangle, \langle l_{j1}\rangle\}$;
3. $EXT_w = \{\langle r = j0\rangle, \langle r = j1\rangle, \langle r! = j0\rangle, \langle r! = j1\rangle\}$;
4. $T_{a_w} = \{wr_{i0}, wr_{i1}\}$;
5. $T_{c_w} = \{\lambda_{i0j0}, \lambda_{i0j0'}, \lambda_{i1j0}, \lambda_{i1j0'}, \lambda_{i0j1}, \lambda_{i0j1'}, \lambda_{i1j1}, \lambda_{i1j1'}\}$;
6. $M_{a_w} = \{(wr_{i0}, (i, 0)), (wr_{i1}, (i, 1))\}$;
7. $M_{c_w} = \{(\lambda_{i0j0}, (i, 0), (j, 0)), (\lambda_{i0j0'}, (i, 0), (j, 0)), (\lambda_{i1j0}, (i, 1), (j, 0)),$
   $(\lambda_{i1j0'}, (i, 1), (j, 0)), (\lambda_{i0j1}, (i, 0), (j, 1)), (\lambda_{i0j1'}, (i, 0), (j, 1)),$
   $(\lambda_{i1j1}, (i, 1), (j, 1)), (\lambda_{i1j1'}, (i, 1), (j, 1))\}$.

### 4.4.2 The reader module

In an overwriting ACM the reader first advances to the next cell and then performs the data access. Again, the next cell depends on the state of the writer. Suppose that the reader is addressing the pair of cell and slot $(i, s)$, with $i = 0..n-1$ and $s = 0, 1$ at a certain time. If the writer has not engaged into an overwriting cycle and is addressing the $j^{th}$ cell, where $j = i + 1 \mod n$, then the reader will next address $(j, s')$, where $s'$ is the last slot addressed by the writer when it was accessing $j^{th}$ cell. In other words, it depends on the tokens in places $l_{j0}$ and $l_{j1}$ of the writer. Note that if the writer is pointing to the $j^{th}$ cell and no overwrite has been done, then the reader prepares to re-read.

The question now is how the reader detects if the writer has started an overwriting cycle or not. This can be easily achieved testing the place $l_{i\overline{s}}$, supposing the reader is accessing $(i, s)$. When the reader advanced to $(i, s)$, it was because the slot $s$ of cell $i$ was holding the oldest non-read data item, which is indicated by the writer with a token on place $l_{is}$. Then, if a token appears on $l_{i\overline{s}}$ while the reader is pointing at $(i, s)$, it is a sign that overwrite has occurred. Note that above we used the notation $\overline{s}$ to denote the opposite of $s$.

Once the reader has detected that the writer is engaged in overwriting, it advances to the next cell until it finds the cell $j$ that the writer is pointing to. Finally, the reader points to the proper slot of the cell $j + 1 \mod n$.

Observe that with this strategy of alternating the slot accessed in each cell, both processes always attempt to address different slots when they are about to access the same cell, which is enough to guarantee data coherence.

Despite of this complexity, the basic behavior of each module used in building the reader process for RRBB ACMs is same as it is for RRBB ACMs: choose the next cell, and then read its content. In Figure 4.10 this basic behavior is captured as a Petri net model in which the data access action are modeled by the transitions $rd_{i0}$ and $rd_{i1}$, and all control actions are abstracted by the transition $\mu_i$.

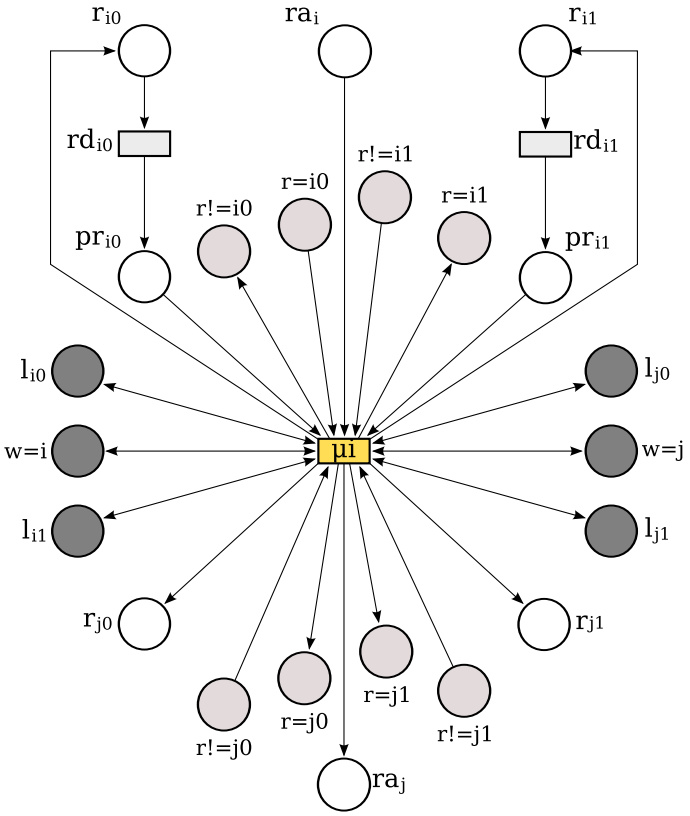
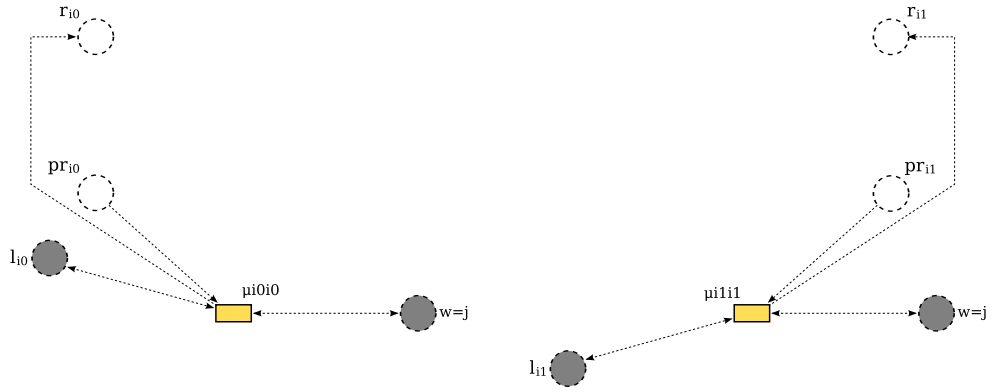



Figure 4.10: The reader module with compressed control actions

The places $r_{i0}$, $r_{i1}$, $pr_{i0}$, $pr_{i1}$, $r_{j0}$ and $r_{j1}$ are internal to the reader process, and they function as a program counter indicating the next action to be performed by the process. $ra_i$ and $ra_j$ are also internal to the reader; they also work as program counters, but they are used only when overwriting is detected and the reader advances until the proper cell. Places $r = i0$, $r! = i0$, $r = i1$, $r! = i1$, $r = j0$, $r! = j0$, $r = j1$, $r! = j1$ model the reader's control variables. They are shared with the writer process in order to communicate

the status of the reader. Finally, the places $w = 1$, $w = j$, $l_{i0}$, $l_{i1}$, $l_{j0}$ and $l_{j1}$ model the control variables of the writer; they can only be tested by the reader.

In Figures 4.11(a) and 4.11(b) the $\mu$ transitions modeling the reader preparing to re-read are shown. In both cases, the writer is pointing to the next cell, i.e. to the $j^{th} = i + 1 \mod n$ cell, and it has not engaged in an overwriting cycle, which is detected by the existence of a token on place $l_{i0}$ when the reader has just accessed $(i, 0)$ or by a token in $l_{i1}$ when the reader has just accessed $(i, 1)$.

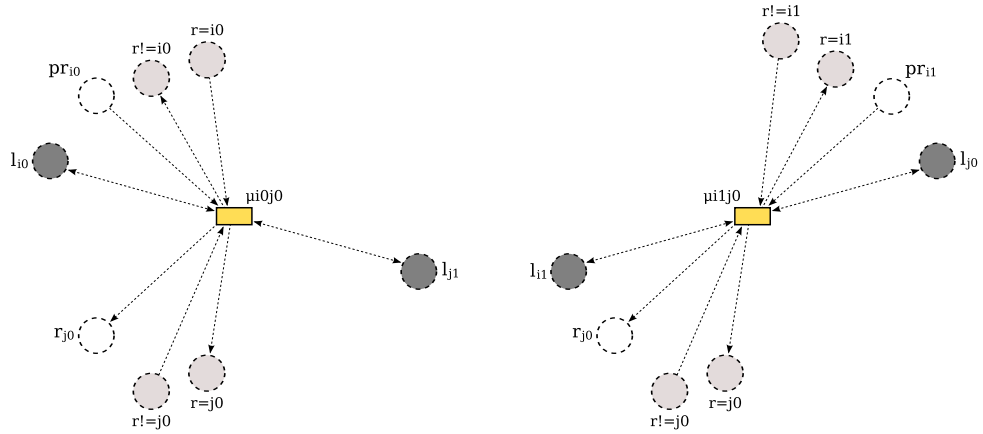



(a) $\mu_{i0i0}$ (b) $\mu_{i1i1}$

Figure 4.11: Control actions for the reader – part 1

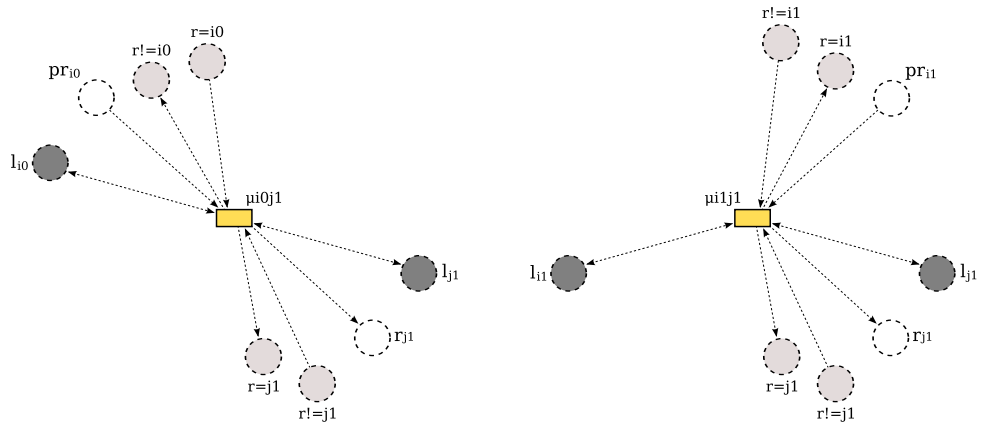
In Figures 4.12 and 4.13 the transitions modeling the reader advancing to the next cell without the occurrence of overwriting are detailed. In all four cases, the writer is not pointing to the $j^{th}$ cell, which is detected by the existence of a token on the place $l_{j0}$ or on the place $l_{j1}$, and also determine the next cell to be pointed by the reader. Besides that, token on place $l_{i0}$ or on place $l_{i1}$ when the reader has just accessed $(i, 0)$ or $(i, 1)$, respectively, indicates that overwriting has not occurred. The shared control variables of the reader are properly updated and so are the internal variables.

Observe that there are four possibilities to be covered: i) moving from $(i, 0)$ to $(j, 0)$; ii) moving from $(i, 1)$ to $(j, 0)$; iii) moving from $(i, 0)$ to $(j, 1)$; and iv) moving from $(i, 1)$ to $(j, 1)$. Each of these possibilities are modeled in Figures 4.12(a), 4.12(b), 4.13(a) and 4.13(b) respectively.




(a) $\mu_{i0j0}$ (b) $\mu_{i1j0}$

Figure 4.12: Control actions for the reader – part 2

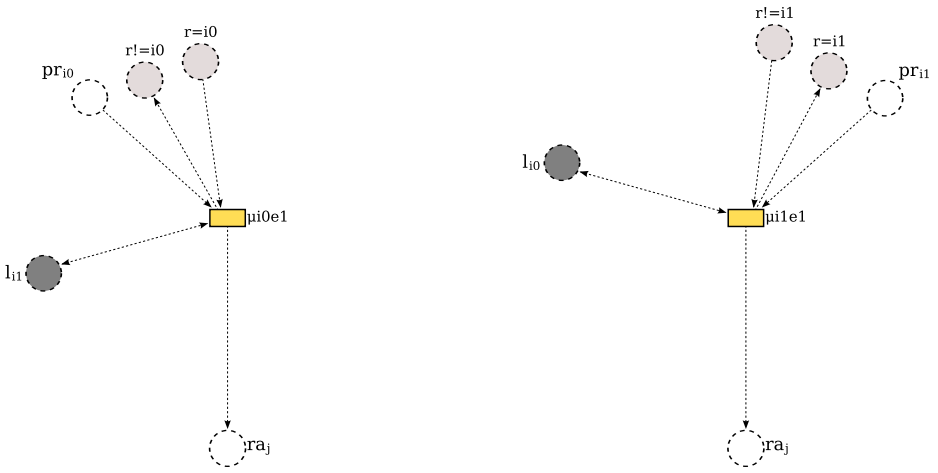



(a) $\mu_{i0j1}$ (b) $\mu_{i1j1}$

Figure 4.13: Control actions for the reader – part 3

In Figures 4.14– 4.16 the transitions modeling the reader detecting an overwriting cycle and running through the shared memory until it finds the

right cell are detailed. In Figure 4.14(a) the transition $\mu i0e1$ models the reader pointing to $(i, 0)$ and detecting the writer engaged in an overwriting cycle by testing the place $l_{i1}$. Remember that a token in $l_{i1}$ when the reader is pointing to $(i, 0)$ indicates that the writer has written a new data item into the $i^{th}$ cell when the reader was accessing it. Also remember that these data accesses are performed in different slots, preserving data coherence requirements. Figure 4.14(b) the same behavior is modeled, but with the reader initially pointing to $(i, 1)$.



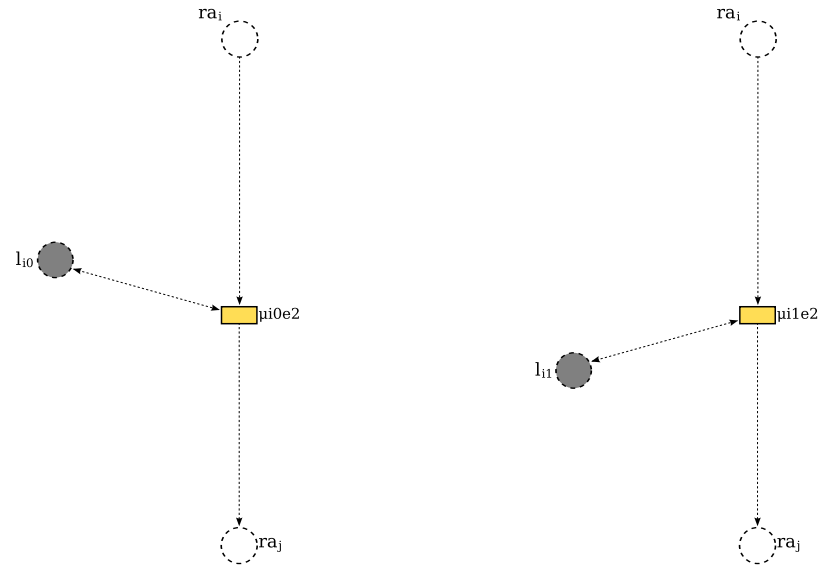

(a) $\mu_{i0e1}$ (b) $\mu_{i1e1}$

Figure 4.14: Control actions for the reader – part 4

After detecting the overwriting cycle the reader enters into a special mode, putting a token on place $ra_j$, and keeps advancing to the next cell until it finds the cell that is being addressed by the writer. When it happens, the reader only tests if the writer is pointing to the cell without preparing to access it. This behavior is modeled by the Petri nets in Figures 4.15.

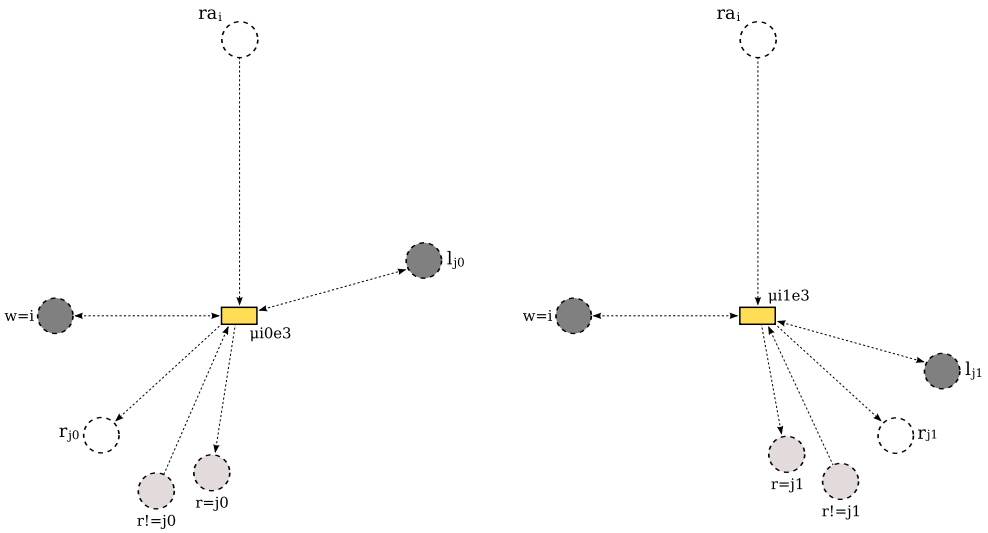
The transition $\mu_{i0e2}$ models the reader testing slot 0 at cell $i$ and finding a new data item there. In this case it advances one more cell in order to test it. The same behavior applies to transition $\mu_{i1e2}$, but in this case the reader is testing slot 1 at the same cell. These transitions are modeled by the Petri nets of Figures 4.15(a) and 4.15(b) respectively. The reader process presents this behavior until it finds the cell which the writer is pointing to.

In Figure 4.16 the Petri net modeling the reader finding the cell holding the oldest non-read data item is modeled by transitions $\mu_{i0e3}$ and $\mu_{i1e3}$. The




(a) $\mu_{i0e2}$ (b) $\mu_{i1e2}$

Figure 4.15: Control actions for the reader – part 5




(a) $\mu_{i0e3}$ (b) $\mu_{i1e3}$

Figure 4.16: Control actions for the reader – part 6

proper cell $j$, with $j = i + 1 \mod n$, is found when: i) the reader finds the cell $i$ such that there is a token in place $w = 1$ indicating that the writer is pointing to it; ii) such $i^{th}$ cell has no indication of a new item, i.e. $l_{i0}$ and $l_{i1}$

have no tokens, avoiding a conflict with one of the transitions $\mu_{i0e2}$ or $\mu_{i1e2}$; and iii) the reader finds a new non-read data item on the $j_{th}$ cell which is indicated by a token in $l_{j0}$ or in $l_{j1}$. Depending on which slot the data is found, the $\mu_{i0e3}$ or $\mu_{i1e3}$ will be fired and the reader will prepare to read the proper slot.

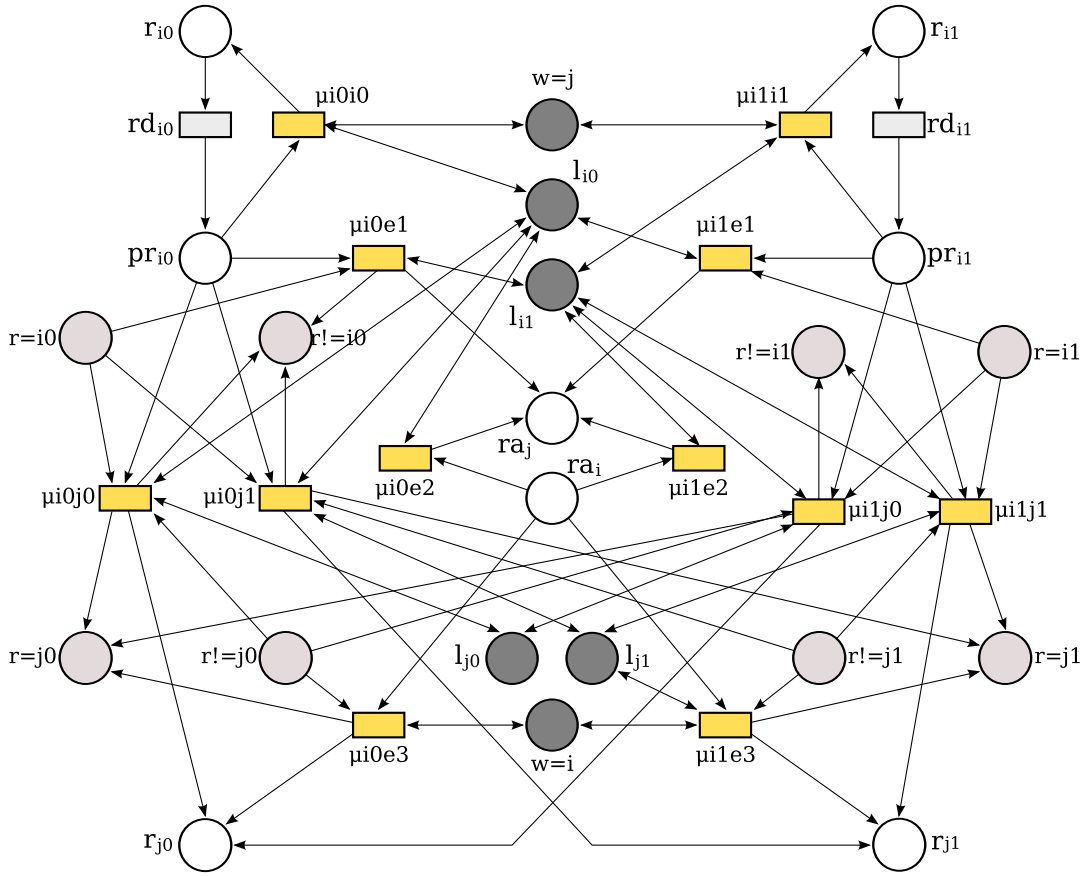



Figure 4.17: The complete reader module

Finally, all $\mu$ transitions are combined to replace the abstraction $\mu$ that appears on Figure 4.10. After replacing it, the Petri net model shown in Figure 4.17 is obtained. The final result is even more complex than it is for the writer. Building and combining both processes result in a Petri net model that is too complex to be handled by a human designer. Also, building those models by hand is a procedure prone to errors. For this reason, a formal definition of the modules for building the writer and the reader processes are needed. Also, a procedure that can be executed by a computer program is essential to obtain the models of such processes without running the risk of

errors.

The Petri net module for the reader process of an OWRRBB ACM is formally introduced in Definition 4.10.

**Definition 4.10 (OWRRBB reader module)** *The OWRRBB reader module is a tuple* $READER = (PN_r, LOC_r, EXT_r, T_{a_r}, T_{c_r}, M_{a_r}, M_{c_r})$ *where:*

1. $PN_r$ *is as defined by Figure 4.9;*
2. $LOC_r = \{\langle r = i0\rangle, \langle r! = i0\rangle, \langle r = i1\rangle, \langle r! = i1\rangle, \langle r = j0\rangle, \langle r! = j0\rangle, \langle r = j1\rangle, \langle r! = j1\rangle;$
3. $EXT_r = \{\langle w = i\rangle, \langle w = j\rangle, \langle l_{i0}\rangle, \langle l_{i1}\rangle, \langle l_{j0}\rangle, \langle l_{j1}\rangle\};$
4. $T_{a_r} = \{rd_{i0}, rd_{i1}\};$
5. $T_{c_r} = \{\mu_{i0j0}, \mu_{i0j1}, \mu_{i1j0}, \mu_{i1j1}, \mu_{i0e1}, \mu_{i0e2}, \mu_{i0e3}, \mu_{i1e1}, \mu_{i1e2}, \mu_{i1e3}, \};$
6. $M_{a_r} = \{(wr_{i0}, (i, 0)), (wr_{i1}, (i, 1))\};$
7. $M_{c_r} = \{(\mu_{i0j0}, (i, 0), (j, 0)), (\mu_{i0j1}, (i, 0), (j, 1)), (\mu_{i1j0}, (i, 1), (j, 0)), (\mu_{i1j1}, (i, 1), (j, 1)), (\mu_{i0e1}, (i, 0), (j, -1)), (\mu_{i0e2}, (i, -1), (j, -1)), (\mu_{i0e3}, (i, -1), (j, 0)), (\mu_{i1e1}, (i, 1), (j, -1)), (\mu_{i1e2}, (i, -1), (j, -1)), (\mu_{i1e3}, (i, -1), (j, 1))\}.$

### 4.4.3 Connecting modules for OWRRBB ACMs

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 4.11 captures this.

**Definition 4.11 (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* $F = F_1 \bigcup F_2 p$

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 4.11 except that rules 2 and 3 do not apply.

### 4.4.4 Initial marking for OWRRBB ACMs

The last required step is to set an appropriated initial marking for the Petri net model. This can be done using Definition 4.12.

**Definition 4.12 (Initial marking for OWRRBB ACMs)** *For any Petri net model of an RRBB ACM, its initial marking is defined as follows.*

1. $M_0(w_{11}) = 1$;
2. $M_0(w = 1) = 1$;
3. $M_0(l_{i1}) = 1$, *if* $i \neq 1$;
4. $M_0(r_{01}) = 1$;
5. $M_0(r! = i0) = 1 \quad \forall i = 0 \cdots n-1$;
6. $M_0(r = 01) = 1$;
7. $M_0(r! = i1) = 1 \quad \forall i \neq 0$

*All other places are not marked.*

Observe that according to Definition 4.12, 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.

## 4.5 Conclusions

In this chapter the generation of Petri net models of ACMs using a modular approach has been introduced. Differently to what has been presented in Chapter 3, the method discussed here can be used in practice to obtain ACMs of big sizes, which was not possible in practice with the previous method. The payback is the need for formal verification, which will be discussed on the next chapter, of the Petri net models obtained using the modular approach.

The behavior expected from the re-reading and overwriting ACMs has been defined as a transition system and specified as a set of CTL formulae. Then the generation of Petri net models has been detailed. The basic modules used to obtain the models were defined, together with the procedure used to connect them and build to entire model.

The reader may have noticed that there is no comparison between the Petri net models generated on this chapter with the models obtained with the method presented on Chapter 3. This is mainly because the method presented on Chapter 3 cannot be used in practice. It is possible to obtain only ACMs for the re-reading policy. For the overwriting policies, even very small ACMs cannot be generated. For the particular case of the re-reading ACMs, the obtained Petri nets are too similar. This is not a surprise, since the modular method as designed based on the models obtained with that technique. However, we have made comparisons only for very few and small models, and we cannot ensure this is true for any generated model.

Unfortunately, since the method introduced in Chapter 3 has limited practical use, a comparison between the results of both methods is not possible, in particular for overwriting policies. For the re-reading policy, the Petri net models obtained from both methods are basically the same. It is not a surprise since the modules were designed based on the Petri net models obtained from the first approach. It should be said that the ACM-region based method presents some advantage over the modular one. For some few examples, a smaller Petri net is generated. Smaller means fewer variables, which is good both for code synthesis and verification.

However, even for the smaller overwriting ACM a comparison is not possible. This is mainly because the model obtained is too cumbersome to be handled by a human. Differently to what happen to the re-reading policy, the modules for the overwriting ACMs were designed from scratch without any hint. It is possible the Petrify generates models that optimized compared to the models generated with the modular approach. In this case, the source code generated may also be faster. But this is pure speculation.

Finally, the source code can also be obtained from the state graph specification, instead of sing Petri net models. The disadvantage of this alternative

is that it will be necessary to deal with an explicit representation of the state space of the ACM, while with the Petri net it is not.

# Chapter 5

# Building and verifying ACM models

This chapter will discuss how the verification of ACMs has been performed using the Petri net models obtained through the method described on Chapter 4. The verification of two properties have been addressed: coherence and freshness. Both have been described previously. More specifically, this chapter details how the abstract description of ACMs introduced on Section 4.2 was modeled using the Symbolic Model Verifier (SMV) [51]. Besides that, it is also discussed how CTL formulae specifying coherence and freshness properties for ACMs of a given size are obtained. Then the SMV models and CTL formulae are used to perform model checking in order to guarantee that the properties are satisfied by the models. However, since the ACMs artifacts we intend to obtain are synthesized from the Petri net models and not from those abstract models, it is also necessary to verify the Petri net models. In this work it is done by verifying that the PN model is a refinement of the abstract model. In this way, it is possible to assure that all PN models satisfy coherence and freshness. This is necessary to increase the reliability of the ACM source code which is generated from the PN model. Since the source code mimics the PN behavior, it should also preserve the referred properties.

## 5.1 Overview

The approach used in this work for the verification of ACMs consists of five main steps, as outlined in Figure 5.1. Since the starting point of the ACM synthesis method is a functional specification that consists only of a policy specification and the size of the buffer, it is natural that the first step is the generation of the models and properties to be verified.

In this way, to perform verification it is necessary to generate the set of CTL formulae specifying the desired properties of the ACM, i.e. coherence and freshness, and the models that should be verified against these formulae. The formulae are generated according to Equations 4.1 and 4.2 if the ACM to be verified is a re-reading one, or according to Equations 4.3 and 4.4 of it is an overwriting ACM.

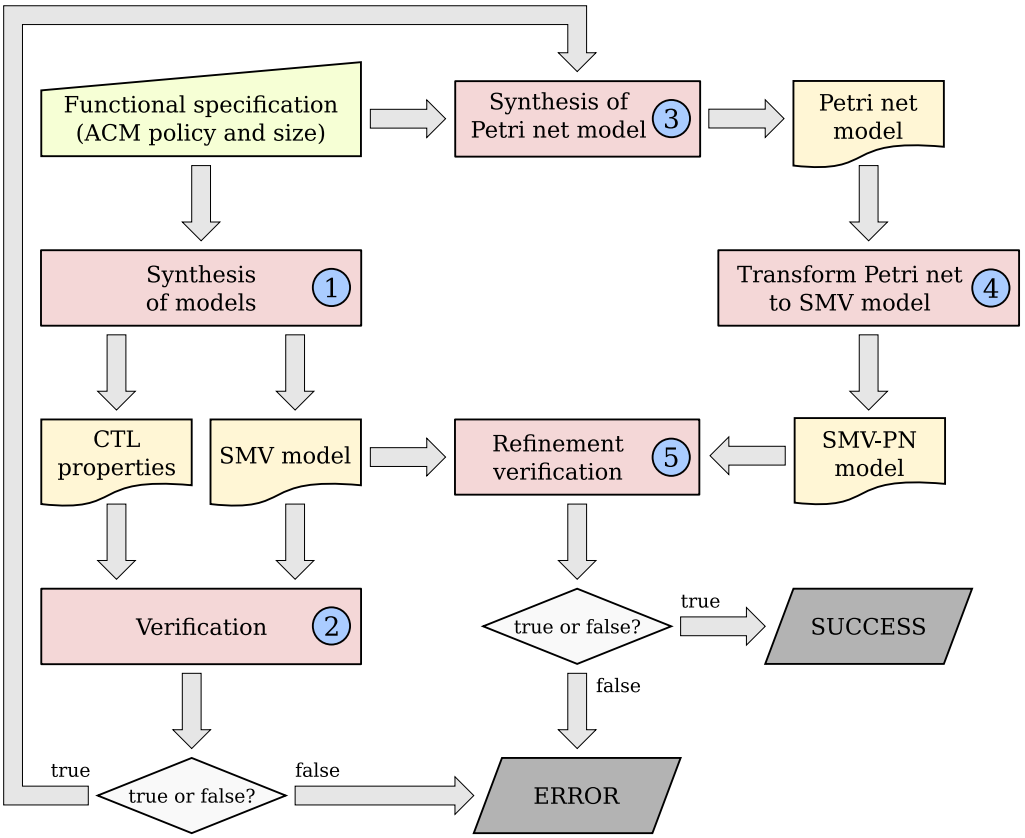



Figure 5.1: The verification flow

Then an SMV model of the ACM is generated according to Definitions 4.1 or 4.2, depending on the policy of the ACM. These two tasks are not part of the verification itself. However, they are required to obtain the models and properties to perform the verification. Without them, verification with SMV is not possible. The model obtained here is called the *abstract model*. Together, these two tasks constitutes the first step of the verification flow outlined in Figure 5.1.

With the CTL formulae and the SMV model *at hand* it is possible to execute the second step of the verification flow, which consists in the first

execution of the SMV model checker. This first run serves the purpose of ensuring that the abstract model satisfies coherence and freshness properties. If they are satisfied, then it is a good start into direction on ACM synthesis. However, the abstract model alone is not appropriated for the synthesis of ACMs implementations. It is too abstract for that purpose. It is necessary to have a more detailed specification. For this, the Petri net model should be used.

The Petri net model, which will be called the *low-level* model, is generated on the third step according to the method presented in Sections 4.3 and 4.4, depending on the type of ACM under synthesis. Once generated, this model needs to be verified to ensure its correctness with respect of the abstract model. For this purpose, refinement verification can be applied.

The refinement verification method consists in verifying if the behavior implemented in a low-level model is correct with respect to the behavior of a more abstract model. In other words, refinement verification will be used here to check if the behavior implemented by the Petri net model is correct with respect to the behavior specified in the SMV model. The first problem that arises is that both models are not expressed in the same language. One is an SMV model while the other is a Petri net. For this reason, the fourth step consists in translating the Petri net model into an SMV one. Since the Petri nets generated are always one-safe, such transformation is possible.

Besides this transformation it is also necessary to apply some modifications to the low-level model in order to specify the refinement relation. Specifically, it is of interest to determine whether, for a certain sequence of data that is transmitted, the reader will recover the correct information. The correct information is given by the abstract model. These modifications will be detailed along this chapter.

Finally, once all models have been generated and the refinement relation has been described, the fifth step, which is the refinement verification, is performed. If the result of this verification is positive, then the ACM source code (hardware or software) can be obtained with the formal guarantee that the artifact will preserve the properties.

The reader will notice that the methodology does not foresee any action if step 2, or 5, fails. As stated before, this is because verification is used as a validation tool of the methodology and not of the protocols. In other words, the user is not expected to perform any verification in the models. Since there is no formal proof of the correctness of the methodology, this validation is required to provide some confidence that the method indeed works.

The ACMs introduced in this work have also been modeled as Coloured Petri Nets [37, 38]. Besides the fact that this activity is not included in the verification flow of Figure 5.1, the results obtained with the CPN models are

also included here for completion. The CPN models have been subjected to verification using the ASKCTL library and a set of Message Sequence Charts (MSCs) have been generated from the simulation of the model [27, 28]. The generation of MSCs is particularly important for a better understood of the ACM policies described here for those unfamiliar with the ACM protocols.

## 5.2 Verification of the abstract models

The first step towards the verification of the ACM models generated with the method proposed in this work is to model the behavior described in Definitions 4.1 and 4.2 using some tool that can perform automatic model checking of these models. Those definitions describe ACMs as finite state machines, in which the state of the system is given by the status of the communication queue. It should be clear by now that those finite state machines are not general for ACMs of any size and that, by consequence, ACMs of different sizes have different finite state machines. For the same reason the set of CTL formulae describing coherence and freshness properties also changes according to the size of the ACM. More specifically, when the size of the ACM grows, new CTL formulae are needed to describe those properties correctly.

The behavior of ACMs has been described in this work through an SMV model that can be parametrized. In order to obtain the SMV model for an ACM of a given size, it is only necessary to set the size parameter properly. This applies to both re-reading and overwriting policies.

The models obtained for both policies use the same schema. The SMV modules feature is used to define three basic modules. The first one models the behavior of the writer process. The second one models the behavior of the reader process. Finally, the third one is the main module, and is used to instantiate one writer and one reader as SMV processes. The main module is also used to instantiate some variables and connect both processes. In what follows, these SMV models will be detailed.

### 5.2.1 SMV models for re-reading ACMs

As stated above, the SMV model for RRBB ACMs consists of three SMV modules: one for the writer, one for the reader, and one to instantiate and connect them, as shown in Figure 5.2. The model can be parametrized according to the size of the ACM.

In the SMV language a module consists of a set of definitions that can be reused. Modules may have parameters that can be used to connect the

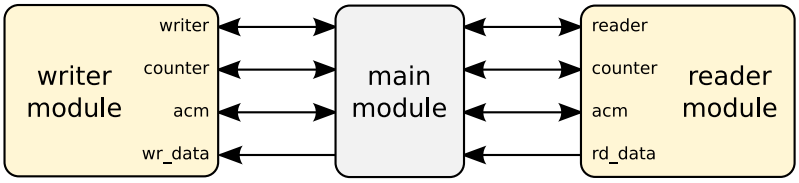



Figure 5.2: ACM SMV modules

instance of a certain module to the rest of the model. In some sense, an SMV module is similar to a process definition, and the entrance point of the model is the `main` module. In the ACMs models defined in this work the main module is used to instantiate the writer and reader processes, which are also modeled as modules, and connect them properly. Besides that, the variables defining the ACM according to Definition 4.1 are declared and initialized. For instance, from Definition 4.1 it is very clear that each process can be *idle* or *accessing* the communication queue, and it is necessary to define those possible states. It is also necessary to declare the communication buffer, which can be an array, and the variables that will be written into the buffer and where the recovered information will be stored. Finally, it is necessary to count how many items there are in the buffer at a given instant time.

The variables **writer** and **reader** are used to control the status of the writer and reader modules respectively, and they can assume the values *idle* or *accessing*. **wr_data** is used to indicate which is the new data to be written in the communication buffer, while **rd_data** stores the last data that has been read from it. The amount of data available in the buffer is indicated by variable **counter**, and, finally, **acm** is an array that represents the communication buffer. The array of data **acm** can store at most as many items as specified by the size of the ACM, and from now on the size of the ACM will be referred as **ACM_SIZE**. Besides **acm**, another array called **acm_previous** is also used, it will be necessary in order to verify the freshness property. For now it is enough to know that it has the value **acm** had in the previous state.

When the writer stores a new item in the buffer, **counter** is incremented. When the reader removes one data item from the buffer, **counter** is decremented. Note that **counter** is set by both reader and writer modules. One can think this is a contradiction with the ACM framework in which the control variables are unidirectional. However, it is necessary to take into account that at this point, we are only modeling the behavior specified in Definition 4.1, which does not impose any constraints about how control variables can be used.

The next step is to initialize those variables. The status of both processes is set to *idle*, the buffer is initialized with some data, and the data counter is set to one.

### Defining SMV modules

The last step towards the modeling of the abstract behavior of RRBB ACMs is the definition of the behavior of the writer and reader modules. This can be easily achieved by defining a Finite State Machine (FSM) for each module. In Figures 5.3 and 5.4 the FSMs representing the behavior of the writer and reader modules are introduced.

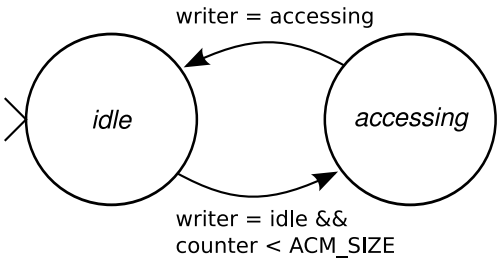



Figure 5.3: FSM for the SMV writer module (RRBB)

Initially the writer is in the *idle* state. If there is some empty cell in the buffer, which is indicated by the condition **counter** $<$ **ACM_SIZE**, then the writer can proceed and start writing a new data item. On the other hand, if the writer is accessing the buffer, it can release the new data item at any time. These are in accordance with Rules 1 and 3 and Definition 4.1 respectively. Note that if none of these is true, then the writer cannot execute, according to Rule 2 of the same Definition.

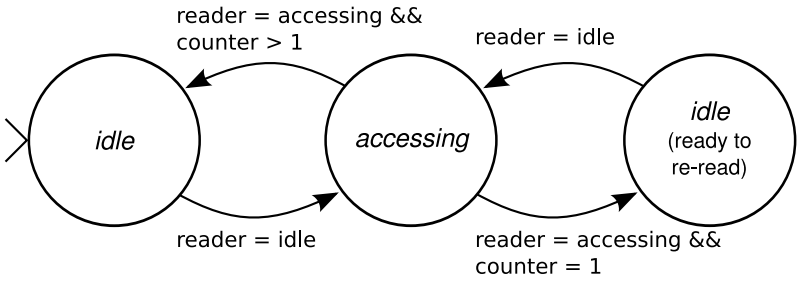



Figure 5.4: FSM for the SMV reader module (RRBB)

On the other hand, the reader is not required to block under any circumstance. Initially the reader is also at *idle* state, and it may start accessing the buffer at any time, which corresponds to Rule 4 of Definition 4.1. If the reader is already accessing the buffer two things may happen after it finishes the data access. First, if there is only one item available in the buffer, it cannot be removed, and in this case the reader prepares to re-read. This is indicated condition **counter=1** and by the *idle* state on the right side which is marked as **ready to re-read**. Since the writer only increments `counter` when releasing the item for reading, there is no difference in the FSM between Rules 5 and 6. Finally, if there is more than one data item available for reading, i.e. **counter** $>$ **1**, then the reader finishes accessing the buffer and removes the current data item from it, returning to the *idle* state; this corresponds to Rule 7.

To better explain how the variables of the model are manipulated, it is necessary to look deep into how the variables change when there is a state transition. For instance, let us look at what happens when the reader changes from *accessing* to *idle* removing one data item from the buffer. This is specified by an SMV code such as:

```
1 next(reader) := idle;
2 next(counter) := counter - 1;
3 for (i = 0; i < ACM_SIZE-1; i = i + 1) {
4     next(acm[i]) := acm[i + 1];
5 }
6 next(acm_previous) := acm;
```

In each line, the value of a variable in the next state is defined in terms of its current value. For instance, since the code above refers to the reader, the status of the writer does not change at all. In line 1 it is specified that the next status of the reader will be `idle`. In line 2 the value of **counter** is decremented by one. In the **for** loop, the data item just read is removed from the communication buffer. This is done by shifting all $i^{th}$ elements of the array **acm** to the previous one, i.e., making `acm[i]=acm[i+1]` for all $i \geq 0$. Finally, the current status of the buffer **acm** is saved in the auxiliary array **acm_previous** for verification purposes.

The rest of the model will not be discussed in details. All the actions specified in the SMV language are intended to implement the behavior described by Definition 4.1. However, in Section A.1 of Appendix A a complete SMV file describing a 3 cell RRBB ACM can be found.

### Generating the CTL specification

The last step towards verification is the generation of CTL formulae as defined by Equations 4.1 and 4.2 on Section 4.2.1. For the coherence property, only one CTL formula is necessary, and it consists only of a syntactic translation from the notation of Equation 4.1, which for the sake of readability is reproduced below, to the SMV syntax.

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

The equation above is described by the following SMV piece of code:

```
SPEC AG(reader = accessing -> (rd_data = acm[0] & counter > 0));
```

where `reader = accessing` corresponds to $a^r \in \sigma$, `rd_data = acm[0]` corresponds to $a^r = a_0$ and `counter>0` simplifies $a^w \in \sigma \rightarrow a^w = a_j \wedge j > 0$. This can be done because the writer always accesses the first free position in the array `acm`, i.e. the first position after the last data item, which is indicated by the value of `counter`. Then it is only necessary to guarantee that `counter` is greater than zero.

Generating CTL formulae for freshness is more complicated because the number of formulae needed depends on the size of the ACM. For the smallest case of a re-reading ACM with three cells, the following CTL formulae are generated:

```
1  -- sequencing properties for counter = 1
2  SPEC AG(counter = 1 ->
3          AX((counter >= 1 & acm[0] = acm_previous[0])));
4
5  -- sequencing properties for counter = 2
6  SPEC AG(counter = 2 ->
7          AX((counter >= 2 & acm[0..1] = acm_previous[0..1]) ||
8             (counter = 1  & acm[0] = acm_previous[1])));
9
10 -- sequencing properties for counter = 3
11 SPEC AG(counter = 3 ->
12         AX((counter >= 3 & acm[0..2] = acm_previous[0..2]) ||
13            (counter = 2  & acm[0..1] = acm_previous[1..2])));
```

In the above, each CTL formula corresponds to the specification of freshness for a given number of data items available in the buffer. More specifically, the first formula specifies that if there is one data item in the buffer, then in the next state at least this item will be in the buffer. The second formula specifies that if the buffer has two data items, then next time it will have the same items or it will have only one item and this item will be the second one in the buffer. Note that in the formulae `acm` refers to the current status of the buffer, while `acm_previous` refers to the status of the buffer on

the previous state. Remember that in the SMV model, it is always backup on each state transition.

Finally, the third CTL formula specifies that if the buffer has three data items, then next it will have at least the same items or it will have two items that will corresponds to the tail of the buffer. Here, the terminology *tail* denotes the last $n-1$ elements of a list of size $n$. If the list is empty, then its tail is also an empty list.

Observe that if the size of the ACM grows to four, then a fourth CTL formula is needed to complete the specification of freshness. This new formula should be similar to the third one, but with some modification on the values the variables should assume.

Let us take the third formula in order to see how the formulae above relate to Equation 4.2. Again, in order to help the reader, that equation is reproduced below.

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

It is easy to note that the basic structure of the formula is just a syntactic translation of $\texttt{AG}(\cdots \rightarrow \texttt{AX}((\cdots \wedge \cdots) \vee (\cdots \wedge \cdots)))$ into `SPEC AG(...->AX((...&...) || (...&...)))`. It is also easy to observe that all atomic propositions involving $|\sigma|$ is related to the variable `counter`. For instance, $|\sigma| = x$ corresponds to `counter=3`, and the same reasoning applies to the other propositions where $|\sigma|$ appears.

The most complicated part is how to express the $\sigma'$, $\sigma^{+}$ and $\sigma^{-}$ in the SMV language. It should be clear by now that at each state of the system, the array `acm` has the data items available for reading in the current state, while `acm_previous` has data values that were available in the previous state. Note that in the `AX(...)` sub-formula all references are made having as starting point a possible next state. $\sigma'$ represents the current status of the data queue, i.e. `acm`, and $\sigma^{+}$ and $\sigma^{-}$ both represent the previous status of the data queue, i.e. `acm_previous`. The difference between them is that $\sigma^{+}$ represents a possible growth on the data queue and $\sigma^{-}$ represents a reduction on it.

In this way, it is easy to map $\sigma' = \sigma^{+}$ into `acm[0..2] = acm_previous[0..2]`, which means that the first three data elements remain unmodified. In this particular case, three is the maximum size of the ACM, and it is not possible to increase the size of the queue. But if it was the case, we just do not care about the status of fourth element in this particular formula. On the other hand, $\sigma' = \sigma^{-}$ is mapped into `acm[0..1] = acm_previous[1..2]`, meaning that the head of the queue was removed and all its tail is shifted to the beginning of the array.

**Results of the generation of SMV models**

Using this schema, we have specified the sequencing determined by the freshness properties using a set of CTL formulae. If all of them are evaluated to true, then freshness is said to be satisfied.

Using the schema described above, RRBB ACMs of size up to 8192 cells have been generated. The generation of the SMV models consists only of changing one parameter in the SMV model describing the ACM. On the other hand, since the number of formulae required to describe freshness grows with the size of the ACM, the time required to generate such formulae also depends on the size of the ACM. The generation of this 8192 cells RRBB ACM took about one hour.

Using an Intel®Core$^{TM}$2 Duo CPU (T5500) at 1.66 GHz with 2GB of RAM, model checking has been performed on a number of those models. It was possible to verify coherence on models up to 1024 cells and freshness on models up to 512 cells. When trying to verify an ACM with more cells, there is not enough memory to store the state space. Table 5.1 summarizes the data collected from the SMV execution. For verifying coherence and freshness of a 512 cells ACM it was necessary 595.94 seconds, and 1089.41 seconds respectively. For the coherence property it was possible to verify ACMs up to 1024 cells, while for freshness the limit was 512 cells.

### 5.2.2 SMV models for overwriting ACMs

The SMV model for overwriting ACMs is similar to the model for re-reading ACMS. In fact they share the same basic structure. It is divided in three SMV modules, one modeling the writer behavior, one modeling the reader, and the third is the main entry point of the model. In the main module, variables are declared, and the processes are instantiated. As in the RRBB model, the model is specific for an ACM of a given size, which can be parametrized.

The main difference between both models is the behavior specified for each process. In what follows, it will be discussed how the rules of Definition 4.2 are related to the SMV model of an OWRRBB ACM. Let us start looking at the writer module, which is given by the FSM of Figure 5.5.

As previously, each module consists of an FSM in which each state change corresponds to a Rule on Definition 4.2. Initially, the writer is *idle*. If the writer starts accessing the buffer and there is some available space, which is given by condition **counter** $<$ **ACM_SIZE**, then the writer changes to the *accessing* state. This corresponds to the Rule 1.

In the case when the buffer is already full of items, the writer engages into an overwriting cycle. There are two possibilities, and in both the writer

| ACM size | user time (coherence) | user time (freshness) |
|---|---|---|
| 3 | 0.02 s | 0.03 s |
| 4 | 0.02 s | 0.04 s |
| 5 | 0.02 s | 0.04 s |
| 6 | 0.03 s | 0.03 s |
| 7 | 0.03 s | 0.06 s |
| 8 | 0.03 s | 0.07 s |
| 16 | 0.07 s | 0.15 s |
| 32 | 0.14 s | 0.48 s |
| 64 | 0.24 s | 2.27 s |
| 128 | 1.07 s | 16.16 s |
| 256 | 8.18 s | 106.03 s |
| 512 | 73.07 s | 1172.65 s |
| 1024 | 602.1 s | timeout |
| 2048 | timeout | |

Table 5.1: Model checking of RRBB ACMs

starts overwriting some data item. The first possibility is that the reader is in the *idle* state, and it corresponds to Rule 2. In this case, the first data item stored in **acm** should be discarded. On the other hand, if the reader is accessing the buffer, the second data item in **acm** will be discarded. This corresponds to Rule 3 of Definition 4.2. Observe that this happens due to the fact that it should not be possible to discard a data item that is being read at the moment. In both cases, the writer should indicate that an overwriting has occurred.

Finally, when the writer finishes accessing the communication buffer, it returns to the *idle* state. This happens at all accessing states, and these transitions correspond to Rule 4.

It can be observed that the rules describing the behavior of the reader process of OWRRBB ACMS are exactly the same that describe the behavior of RRBB ACMS. This is expected, since both sets refer to the re-reading characteristic of the reader. For this reason one can expect that the SMV

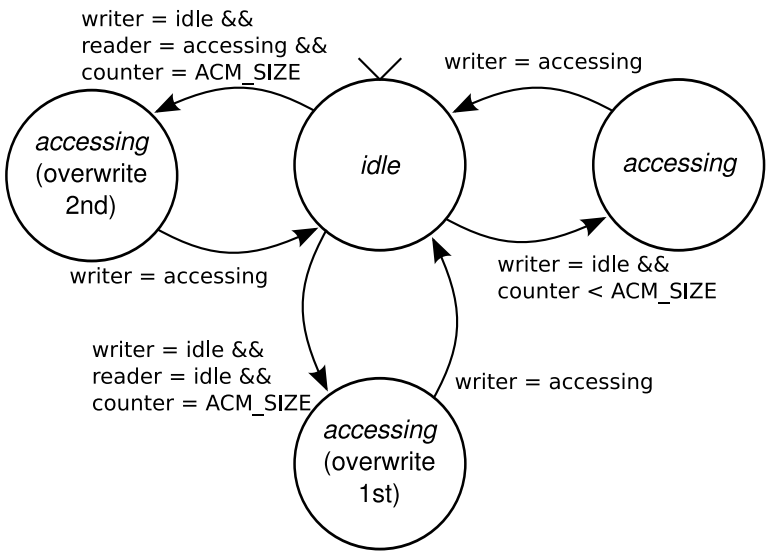



Figure 5.5: FSM for the SMV writer module (OWRRBB)

model for the reader is also the same, and in part this is true because it can be done, and certainly all properties will be satisfied. The problem is that in this case, this model cannot be used to verify if the Petri net model of OWRRBB ACMs obtained by the procedure introduced in Chapter 4 refines it or not.

The reason is that in the Petri net model that is synthesized, the reader needs to *"walk"* through the communication buffer until it finds the memory position it should next access, which does not happens in the conceptual description of Definition 4.2. To solve this problem, a silent action is added to the reader in such a way that it corresponds to that *"walk"*. In other words, this action does not change the values of any variable of the system. This modified FSM is shown on Figure 5.6.

Initially, the reader is *idle*. When it start reading some item, its state changes to *accessing*, which corresponds to Rule 5. Remember that it is not necessary to check the buffer for new items because the reader can always re-read the last one if necessary. Then the reader can finish accessing the buffer and return to *idle*. Again, it can remove a data item from the buffer, or prepare to re-read. As before, removing an item corresponds to Rule 6 while preparing to re-read the item corresponds to Rules 7 and 8 of Definition 4.2.

The *new* silent action is modeled by the arc labeled *overwriting?* at state *accessing*. It represents the reader finishing accessing the buffer but when the writer has engaged into an overwriting cycle. In this case, the reader does

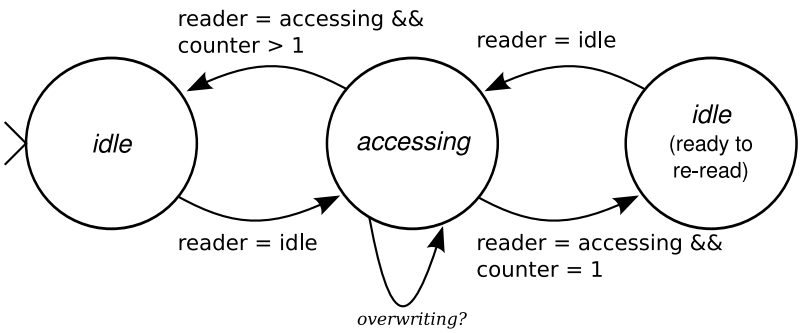



Figure 5.6: FSM for the SMV reader module (OWRRBB)

not modify any variable of the model. At some point the reader should finish the search for the right data item and stop the cycle of silent actions. Again, this is necessary to *synchronize* the SMV abstract model behavior with the more detailed Petri net model behavior when overwriting occurs.

Concerning the generation of the CTL formulae describing coherence and freshness properties, it is done basically in the same way as described in Section 5.2.1. Of course, in this case they are obtained from Equations 4.3 and 4.4. Just for illustration, the set of coherence and freshness CTL formulae for a 2 cells OWRRBB ACM is given by:

```
1  -- COHERENCE
2  SPEC AG(reader = accessing -> (rd_data = acm[0] & counter > 0));
3
4  -- FRESHNESS
5  -- sequencing properties for counter = 1
6  SPEC AG(counter = 1 ->
7          AX((counter >= 1 & acm[0] = acm_previous[0])));
8  -- sequencing properties for counter = 2
9  SPEC AG(counter = 2 ->
10         AX((counter >= 2 & acm[0..1] = acm_previous[0..1]) ||
11            (counter = 1  & acm[0] = acm_previous[1]) ||
12            (counter = 1  & acm[0] = acm_previous[0])));
```

Using the same hardware as before, a number of OWRRBB ACMs models have been generate and verified with the SMV model checker. As before, verification of coherence was feasible for ACMs up to 1024 cells, and 512 cells for freshness. In Table 5.2 the time needed to verify them is summarized.

We omitted here the models of OWBB ACMs, i.e. the policy in which only overwriting is allowed, but it can be easily obtained from the OWRRBB ACM model. In Appendix A complete examples of SMV models for RRBB with 3 cells and OWBB and OWRRBB ACMs with 2 cells each can be found.

| ACM size | user time (coherence) | user time (freshness) |
|---|---|---|
| 2 | 0.02 s | 0.02 s |
| 3 | 0.03 s | 0.04 s |
| 4 | 0.03 s | 0.05 s |
| 5 | 0.04 s | 0.04 s |
| 6 | 0.03 s | 0.06 s |
| 7 | 0.05 s | 0.08 s |
| 8 | 0.04 s | 0.1 s |
| 16 | 0.06 s | 0.22 s |
| 32 | 0.18 s | 0.73 s |
| 64 | 0.55 s | 3.59 s |
| 128 | 3.51 s | 21.12 s |
| 256 | 26.55 s | 136.02 s |
| 512 | 204.49 s | 1196.65 s |
| 1024 | 1685.87 s | timeout |
| 2048 | timeout | |

Table 5.2: Model checking of OWRRBB ACMs

## 5.3 Verification of the Petri net model

The Petri net model generated using the procedure discussed on Chapter 4 will be used to synthesize source code, that can be in C++, Java, Verilog, etc., that implements the behavior specified by the model. In this way, it is necessary to guarantee that such a model is correct with respect to the behavior given by Definitions 4.1 and 4.2 presented in Sections 4.2.1 and 4.2.2 respectively. Up to now, it has been discussed how the SMV model checker was used to verify that SMV models obtained from such definitions satisfy coherence and freshness properties. The problem is that this is not enough to argue that the Petri net models that are synthesized in this work also satisfy such properties, it is necessary to provide some formal guarantee of that.

The specification of a system may be made at different levels of abstractions [3, 11, 50]. This is particularly useful when the system to be verified,

called $S_1$, is too complex that its state space cannot be dealt by the model checker. In this case the designer can build a more abstract model, called $S_2$, of the system and prove the desired properties using this abstract model. Then the designer must demonstrate that $S_1$ is a refinement of $S_2$ with respect to the desired properties, and if $S_2$ satisfies those properties, so does $S_1$. In this case it is said that $S_1$ implements $S_2$.

To achieve this purpose the designer should specify a *refinement mapping* that translates the behavior of $S_2$ into $S_1$ in such a way that every observable behavior of $S_1$ is allowed by $S_2$. The refinement mapping is intended to relate the abstract model behavior to the implementation behavior.

In this work, refinement verification is applied in the following way. The Petri net models that are generated as described in Chapter 4 are used as low-level specifications which are verified against the SMV models described in Section 5.2. Those models are used as the high-level specification for this purpose. Refinement verification is then applied to check if the data items transmitted and recovered in both models are consistent with each other. If this is true, the implementation model is said to satisfy the desired properties.

The first problem to solve is that the high level model has been described as an SMV model, while the low-level model is a Petri net. Since the Petri nets generated are one-safe, i.e. each place has at most one token at a time, its state space is finite. Consequently, it can be easily translated into a finite state machine described using the SMV language. Among other features, the PEP tool [29] supports the translation of a Petri net into an SMV model, and was used the synthesis framework described in this work.

### 5.3.1 Refinement verification

Refinement verification is supported in the SMV by the `layer` statement. In the SMV language the `layer` keyword is used to define a collection of abstract signal definitions. In other words, the definitions inside a `layer` statement are more abstract with respect to the definitions outside the `layer` statement. Its syntax is as follows:

```
1 layer <layer identifier>: {
2
3     <abstract model>
4 }
5
6 <low-level model>
```

In the synthesis framework defined here, the abstract model corresponds to the initialization of the variables and instantiation of the modules as processes (the module definitions themselves are not required to be inside the body of

the `layer`). The low-level model corresponds to the SMV form of the Petri net model.

The Petri net model specifies the mechanisms to control access to the ACM, but it does not model the transfers of real data. Since the goal is to check if the implementation model refines the abstract model by checking the transmitted data, it is necessary to model data transfers in the implementation model. This is done by adding to the implementation model an array for the data, with the same size as the ACM. In other words, after translating the Petri net model to SMV, an array is added to this new model. This new array is called `acmI`, and it has one dimension for re-reading policy and two dimensions for the overwriting policies. This new array is declared in the main module of the abstract model, together with the `acm` array.

Then, the implementation model is modified in the following way to support the data transfer. First, for each event modeling a data access action, it is necessary to add the actions modeling the storage and retrieval of data in the array.

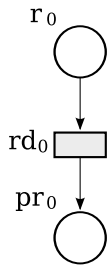



Figure 5.7: Transition $rd_0$ (RRBB)

For instance, the action of reading the item in cell 0, which is given by the Petri net in Figure 5.7, in a three cells RRBB ACM is translated from the Petri net model as the following piece of SMV code:

```
1 --rd0
2   rd0.enabled   &
3       & next(pr0) = 1
4       & next(r0) = 0
```

Observe that the SMV model generated from a Petri net model is an FSM that mimics the behavior of the Petri net. After adding the access to the data array, the SMV code is transformed into:

```
1 --rd0
2   rd0.enabled
3       & next(pr0) = 1
4       & next(r0) = 0
5       & next(rd_data) = acmI[0]
```

```
6       & next(acmI) = acmI
7       & rdp.running = 1
```

Observe that besides the data access in line 5, two more lines were also added. `rd_data` is the only variable in the entire model that is modified by both abstract and implementation models. When performing refinement verification, SMV will understand that this is the variable that defines the refinement relation. In practice it means that if it assumes the same value in both models all the time, then the refinement relation is satisfied.

Line 6 is necessary to avoid non-determinism in the data stored in `acmI`. If it is not present, in the next state the array can assume any possible value. Note that this applies to all variables of the system, and they have been omitted here for the sake of comprehension.

Line 7 specifies a condition that this action will be executed only if the process that will execute next in the abstract model is the reader. It is necessary because at each step, SMV chooses non-deterministically which process will execute next. This is what happens at the more abstract level even if the processes are concurrent as the reader and writer described. However, nothing guarantees that at the low-level an action that corresponds to some action of that process will be also chosen. For this reason, the condition `rdp.running = 1` is added to all events of the reader in the low-level model. The same occurs for the events of the writer, but in this case the condition added is `wrp.running = 1`. Remember that `rdp` and `wrp` are the names given to the instances of the reader and writer modules respectively.

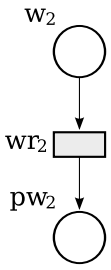



Figure 5.8: Transition $wr_2$ (RRBB)

Just for completion of data access action, which is given by the Petri net in Figure 5.8, the writer is modified to:

```
1  --wr2
2    wr2.enabled
3       & next(pw2) = 1
4       & next(w2) = 0
5       & next(rd_data) = rd_data
6       & next(acmI[0]) = acmI[0]
```

```
7       & next(acmI[1]) = acmI[1]
8       & next(acmI[2]) = wr_data
9       & wrp.running = 1
```

where the last 5 lines have been added to the original model. Observe that each position of `acmI` is set individually.

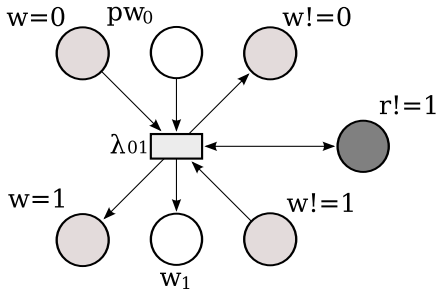



Figure 5.9: Transition $\lambda_{01}$ (RRBB)

On the $\lambda$ and $\mu$ actions, similar modifications are also needed. But in this case they are done only to avoid changes in the values of the variables from one state to another. For instance, the $\lambda_{01}$ given by the Petri net in Figure 5.9, is given by:

```
1   --l0_1
2    l0_1.enabled
3       & next(we0) = 0
4       & next(we1) = 1
5       & next(wne0) = 1
6       & next(wne1) = 0
7       & next(w1) = 1
8       & next(pw0) = 0
9       & next(rne1) = rne1
10      & next(rd_data) = rd_data
11      & next(acmI) = acmI
12      & wrp.running = 1
```

The following steps summarize what should be done to adapt the implementation model in such a way that it can be used in the refinement verification.

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 $i^{th}$ position of the data array created in step 1, where $(t, i) \in M_a$, 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 $i^{th}$ position of the data array created in step 1, where $(t, i) \in M_a$.

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 that 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 with the guarantee that the artifact obtained preserves those properties.

| ACM size | user time (RRBB) | user time (OWRRBB) |
|---|---|---|
| 2 | - | 0.55 s |
| 3 | 0.1 s | 4.61 s |
| 4 | 0.26 s | 652.63 s |
| 5 | 0.73 s | timeout |
| 6 | 2.78 s | |
| 7 | 22.61 s | |
| 8 | 99.9 s | |
| 9 | 301.48 s | |
| 10 | timeout | |

Table 5.3: Refinement verification (RRBB and OWRRBB)

Following the procedure described above a tool to automatically generate ACMs was designed and implemented[1]. A number of RRBBs and OWRRBBs

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

ACMs with different sizes (starting from 2) were generated and proved to be correct for all cases. The time required to verify those models is reported in Table 5.3. For now, it was possible to verify RRBB ACMs with up to 9 cells and OWRRBB ACMs with up to 4 cells using this approach.

## 5.4 A different approach for coherence

One possible problem with the Petri net models generated on this work is that the data access actions are considered as being atomic actions, which is not necessarily true in the implementations. Usually, and especially in hardware implementations, the start of a data access is preceded by a data access request, then the data access proceeds. After finishing the data access, an acknowledgment signal is usually sent to indicate that the operation has terminated.

The Petri net models introduced here can be easily modified to consider data access actions as non-atomic. For this it is only necessary to split each transition that models a data access into two transitions in sequence, i.e. two transitions that are connected by one place that is output of one transition and input of the other. One of the transitions models the start of a data access operation, while the other models the termination if such operation.

The schema of generating the Petri net models can be easily modified to generate such Petri nets. However, one extra place means another variable in the final model, which may make the verification of such systems even more difficult.

In the writer module for RRBB ACMs introduced in Figure 4.2(a) the interpretation of a token on place $w_i$ as the writer is ready to access the $i^{th}$ cells, while a token on place $pw_i$ means that the writer has finished accessing the cell but has not released the item for reading yet. In both cases, the reader is not allowed to access those cells. In the same way, a token on places $r_i$ or $pr_i$ in the reader modules requires the writer not to access the $i^{th}$ cell.

These two conditions together imply that a token on $w_i$ or on $pw_i$ requires the absence of a token on both $r_i$ and $pr_i$. The other way around also applies. Taking this fact into consideration, a possible way for expressing coherence is:

$$\texttt{AG}((w_i \vee pw_i) \rightarrow \neg(r_i \vee pr_i) \wedge (r_i \vee pr_i) \rightarrow \neg(w_i \vee pw_i))$$

Observe that in this case it is not enough to have only one CTL formula to describe coherence. Coherence is described in terms of each cell of the ACM, for this it is necessary to have as many formulae as there are cells. For

instance, to describe coherence for a three cell ACM the following formulae are required:

```
SPEC AG((w0 || pw0) -> !(r0 || pr0) &
        (r0 || pr0) -> !(w0 || pw0))
SPEC AG((w1 || pw1) -> !(r1 || pr1) &
        (r1 || pr1) -> !(w1 || pw1))
SPEC AG((w2 || pw2) -> !(r2 || pr2) &
        (r2 || pr2) -> !(w2 || pw2))
```

In short terms, these formulae specify that at any time the sum of the tokens at places $w_i$, $pw_i$, $r_i$ and $pr_i$ is at most one. Then the alternative coherence formulae can be rewritten as:

$$\mathtt{AG}(M(w_i) + M(pw_i) + M(r_i) + M(pr_i) \leq 1)$$

Generating one CTL formula as described above is sufficient to specify the coherence property. In this case, if coherence is true, then it is also guaranteed that it will also be preserved in the case data access actions are not treated as atomic. This approach has been used to verify with success RRBB ACMs with up to ten cells. In Table 5.4 the time needed to verify the models is shown.

| ACM size | user time | ACM size | user time |
|---|---|---|---|
| 3 | 0.04s | 8 | 3.94s |
| 4 | 0.04s | 9 | 6.09s |
| 5 | 0.1s | 10 | 12.16s |
| 6 | 0.16s | 11 | timeout |
| 7 | 0.99s | | |

Table 5.4: Alternative coherence verification

Observe that the results shown in Table 5.4 differ from the results in Tables 5.1 and 5.3. This happens for two different reasons. In the first case, the results for coherence verification in Table 5.1 are much better due to the fact that they refer to the verification of the abstract model, which is much less complex than the Petri net model or its SMV equivalent.

On the other hand, compared to the refinement verification of RRBB ACMs results shown on Table 5.3, the results above are slightly better. This is because the SMV model used here does not have details about the status of the communication buffer implemented by the variable **acmI**. This variable

is used only for the refinement verification, making the verification more complex.

The need for this alternative approach for the verification of coherence will become more clear on Section 7.2 in which the synthesis of hardware artifacts for ACMs is discussed.

## 5.5 Modeling and validating ACMs with Coloured Petri Nets

In this Section an ACM fitting the behavior introduced in Section 4.2.2 modeled as a Hierarchical Colored Petri Net (HCPN) [37, 38] will be described. Then the properties of data coherence and data freshness will be defined using ASKCTL [6, 7], and the HCPN model of the ACM will be verified for these properties. Besides that, the HCPN models will be used to automatically generate a set of Message Sequence Charts [31] from the simulation of the HCPN model.

As stated in Section 5.1, this verification activity does not fit into the verification flow outlined on Figure 5.1. However, the MSCs obtained from the simulation of the model can be used by those unfamiliar with the topic to have a better comprehension of the behavior of the ACMs.

At this point, one can ask why not use CPN for synthesis directly instead of using a combination of one-safe Petri nets and SMV models. The answer is that it is easier to synthesize hardware, in which variables are mapped into wires, from those kinds of Petri nets than from CPNs. Basically, there is no need to map the rich data types of CPN into the binary world.

### 5.5.1 HCPN models for overwriting ACMs

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 transitions and fusion places. A substitution transition is a transition that represents a CPN page. The fusion places are physically different but logically they are the same, defined by means of a fusion set. All places belonging to the same 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, the concept of a multi-set is defined. 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 5.10 and 5.11 shows the HCPN models for the writer and reader processes as introduced by Definition 4.2, respectively. Each process has two transitions, one modeling the beginning of a buffer access 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.

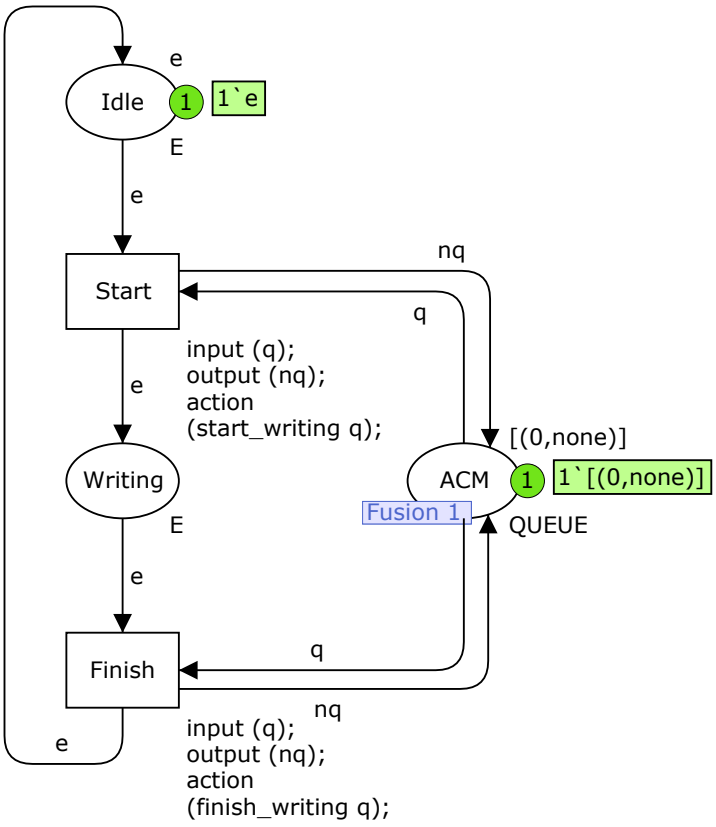



Figure 5.10: CPN model for the writer

Places labeled *ACM* in the page of the writer and in the page of the reader model the queue of data $\sigma$. 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 on. 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 mitigate 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 5.10, 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)]**.

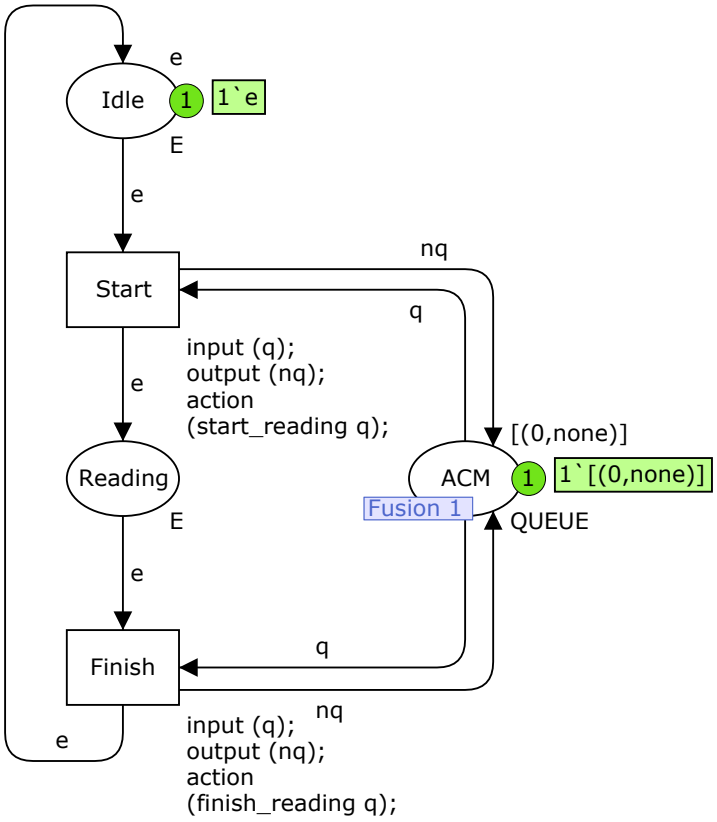



Figure 5.11: CPN model for the reader

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 item 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 the first element of the list of data items can be removed, depending on the status of the queue. 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 any event of the writer. Observe that neither process is required to wait for the other in any situation

### 5.5.2 SML functions

In 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 below shows such a function.

```
1  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 in the queue is removed and the writer starts putting a new data item to the end of the queue (line 10). This code implements the behavior defined by rule 3 in Definition 4.2.

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 the writer starts storing a new data item at the end of the queue (line 13). This implements the behavior of rule 2 of Definition 4.2. Finally, if the queue is not full, the writer simply starts storing a new data at the end of the queue (lines 14 and 15), which implements rule 1 of Definition 4.2.

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 below.

```
1  fun finish_writing data:QUEUE =
2  (
3  let
4    val mesg = "finish_writing";
5  in
6    if (length (tl data) > 0) then
7      [hd data]^^finish_writing(tl data)
8    else (
9      msc.addEvent("writer", "reader", mesg);
10     [(#1 (hd data), none)]
11   )
12 end
13 );
```

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, so it needs to check that the queue is not empty (i.e. the data queue has only its head). `finish_writing` implements the behavior defined by rule 4 of Definition 4.2.

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 4.2.

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 whether 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 starting 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.

### 5.5.3 Generating Message Sequence Charts

In order to illustrate the behavior of the CPN model introduced above a number of Message Sequence Charts (MSC) [31] has been automatically generated from the simulation of the model. For instance, in Figure 5.12 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.

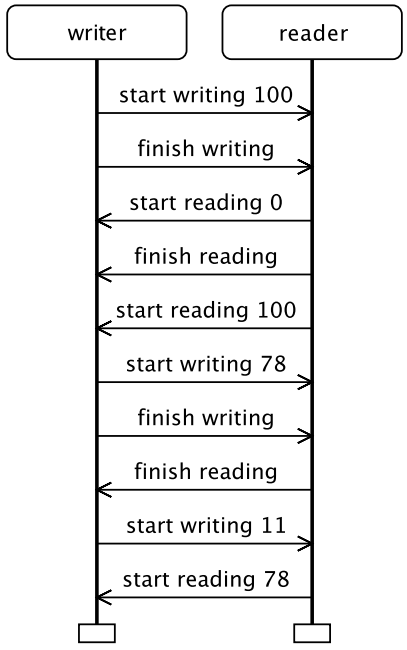



Figure 5.12: MSC for no re-reading and no overwriting case

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 message **finish writing** indicates that the writer is releasing the new data item 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 im-

plementation 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 4.2, 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 $\sigma_i$ 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 5.13 the MSC generated when re-reading occurs is illustrated. In this case reader accesses the buffer twice, recovering the data value **0** on both accesses, before the writer accesses 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 access the reader recovers the new value **100**, otherwise it should engage in another re-reading operation.

For this sequence of messages, the sequence of states according to Definition 4.2 is as follows:

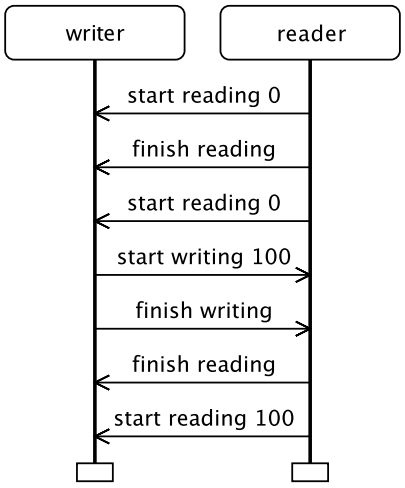



Figure 5.13: MSC for the re-reading case

$$\sigma_0 = 0$$
$$\sigma_1 = 0^r$$
$$\sigma_2 = 0$$
$$\sigma_3 = 0^r$$
$$\sigma_4 = 0^r\ 100^w$$
$$\sigma_5 = 0^r\ 100$$
$$\sigma_6 = 100$$
$$\sigma_7 = 100^r$$

Finally, in Figure 5.14 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 data items at a time. In this case the sequence of states is as follows:

$$\sigma_0 = 0$$
$$\sigma_1 = 0^r$$
$$\sigma_2 = 0^r\ 100^w$$
$$\sigma_3 = 0^r\ 100$$
$$\sigma_4 = 0^r\ 78^w$$
$$\sigma_5 = 0^r\ 78$$
$$\sigma_6 = 78$$
$$\sigma_7 = 78^r$$

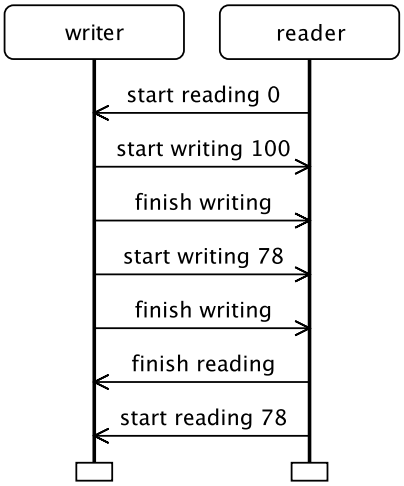



Figure 5.14: MSC for the overwriting case

As can be observed the MSCs generated by the model discussed above reflect the behavior defined by the transition system introduced by Definition 4.2. 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 4.2. Besides that, they promote an intuitive way of understanding how the ACM policies work.

### 5.5.4 Verification with ASKCTL

In order to give formal arguments of the correctness of the model, coherence and freshness properties introduced in Section 4.2.2 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 CPNTools [39, 59]. 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 in 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 $\rightarrow$ is not available, so it is necessary to rewrite the part of the CTL formula for coherence 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 a parameter and returning a Boolean. The following ASKCTL formula is then obtained:

```
1 INV(OR(NOT(NF("reading", has_rd)),
2         AND(NF("reading␣head", rd_first),
3             OR(NOT(NF("writing", has_wr)),
4                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` check 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 refers to a node of the state graph and not to an arc. The formula above is enough to verify ACMs of any size.

Freshness is much more complicated. 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:

```
1  INV(
2    OR(
3      NOT(NF("length␣of␣sigma␣is␣2", check_length 2)),
4      FORALL_NEXT(
5        OR(
6          AND(
7            OR(
8              NF("|sigma'|␣=␣2", check_length 2),
9              NF("|sigma'|␣=␣2+1", check_length 3)
10           ),
11           NF("sigma'␣=␣sigma+", check_sigma_plus)),
```

```
12            AND(
13              NF("|sigma'| = 2-1", check_length 1),
14              NF("sigma' = sigma-", check_sigma_less)
15            )
16          )
17        )
18      )
19  );
```

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 at the form defined by $\sigma^+$, and `check_sigma_less` checks if it is of the form $\sigma^-$ as defined in Section 4.2.2.

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 5.15 this modification is shown.

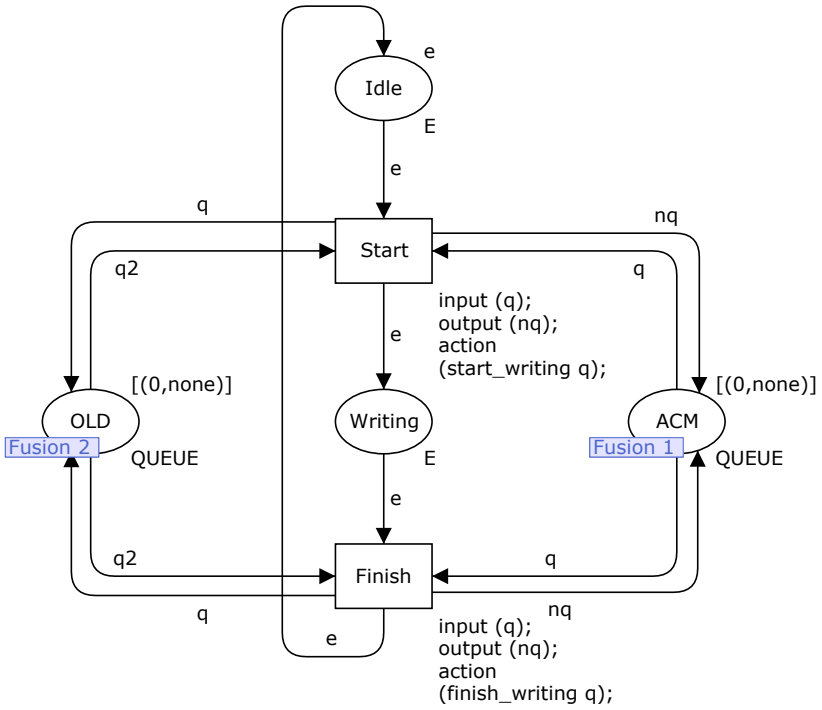



Figure 5.15: CPN model for the writer saving queue state

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.

## 5.6 Conclusions

On this chapter the verification of ACMs models has been introduced. The properties addressed were coherence and freshness. The verification methodology discussed here is mainly based on two steps. Firstly, the ACMs are modeled in a more abstract way, without restricting the models to the use of binary control variables or adding much detail about how the communication buffer is implemented. The abstract models are build in order to correspond to Definitions 4.1 and 4.2. These abstract models are then used to verify coherence and freshness properties. The generation of the CTL formulae needed to perform model checking is also detailed in this chapter.

Secondly, the Petri net models are translated into equivalent SMV models, and mapping relations from the SMV abstract models to the SMV PN-based ones are defined. This allows the designer to perform refinement verification in order to determine if the PN model can be said to be an implementation of the abstract model.

The results obtained for the verification of the abstract model were quite satisfactory, giving some confidence that coherence and freshness may be satisfied for ACMs models of any size. On the other hand, the Petri net models used in the refinement verification are too complex to obtain good results, especially for the OWRRBB policy, and the state space explodes even for very small ACMs. This is an indication that further investigation into direction of getting better verification results is needed. One possibility is to apply proof by induction techniques, and another possibility is to reduce the complexity of the PN model and extend the limits of model checking.

It is important to observe that the verification methodology introduced here is mainly based on the transformation of models from one language to another. This translation process has not been proved to be free of errors, which constitutes a weakness in the methodology. The correctness of the PN models with respect to the abstract SMV models also depends on the correct implementation of the procedures performing the transformation of the PN

models into SMV ones.

Besides that, the ACMs have also been modeled as HCPN models. This model has also been verified in order to prove coherence and freshness properties. However, one of the most interesting results obtained from the HCPN models was the possibility of automatically generating MSCs in which the label of the events are the data access actions. This is very useful to better understand how the ACM protocols should behave.

# Chapter 6

# Automatic synthesis of ACM software implementations

The main goal of this work is to provide a fully automatic method for the synthesis of ACMs. The synthesis procedure should guarantee that the artifact generated satisfies some desired properties. Up to now, the generation of formal models of ACMs was detailed. This chapter introduces how to synthesize an ACM implementation from those formal models. More specifically, the synthesis of C++ [19, 20, 74] source code for both re-reading and overwriting policies that implements the behavior specified in the Petri net model is addressed.

## 6.1 An historical perspective on code generation

Code generation from Petri net models has been studied since the end of the seventies. Many approaches have been proposed for code generation. Addressing a wide range of target languages, from machine code to high level languages, such as C, Java or Ada. Such approaches are classified into three categories:

1. Centralized;
2. Totally decentralized;
3. Hybrid.

The centralized approaches attempts to check each transition of the model in order to determine if it can be fired in the current state. There are efficient

procedures that minimize the number of transitions evaluated at a time, but this approach has two main drawbacks. Firstly, the efficiency of such approach depends on the size of the net. Even optimized algorithms may not have a satisfactory performance when the net grows. Secondly, such procedures are sequential, which does not preserve the parallelism of the models. On the other hand, for systems that are essentially sequential, such approach can be considered adequate.

Silva [64] details two centralized approaches for code generation of binary Petri nets. In the first approach, a Petri net model is simulated through a Programmable Logic Controllers (PLC). A PLC is a digital computer used for automation of electromechanical processes, such as the control of production lines in the industry. In the second approach, the Petri net model is simulated trough assembly code of a general purpose microprocessor, the Motorola 6801.

In the totally decentralized approach, each place and transition of the Petri net is implemented as a process. In this case the problem of preserving the parallelism of the Petri net is solved satisfactorily. Such approach requires proper primitives to communicate the processes. Traditional IPC methods, such as semaphores and monitors, can be used for this purpose. The drawback is that the number of processes grows according to the size of the net. The large number of processes, associated to the overhead introduced by the IPC methods, may result in implementations with poor performance. Taubner [76] used this approach to generate Occam source code from P/T nets, while Hauschildt [32] addressed the generation of C source code for Unix operating system from P/T nets.

The hybrid approach attempts to obtain high level information about the Petri net model, partitioning it into sub-models that are mapped to communicating processes. Kordon [43] presents a hybrid approach to manual partitioning the Petri net model into communicating processes. It is focused on P/T nets and on the generation of Ada source code that can be compiled.

There is a large number of works that address the code generation from Petri net models problems. Most of them deal with code generation with the purpose of obtaining a prototype that can be used to evaluate characteristics that cannot be evaluated in the formal model, such as the performance of a given implementation strategy. It is also possible to generate code from high level Petri nets, such as Coloured Petri Nets. Mortensen [53] outlines a methodology for code generation from CPNs, which is supported by the Design/CPN tool, and demonstrates its usability through an industrial application example. Such methodology can be used to generate C, Java, or even machine code. A much more comprehensive discussion about code generation from Petri nets can be found in [23] and the interested reader is encouraged

to consult them.

## 6.2 Overview

In this work the author does not intend to introduce a new method for code generation. The method described here is much like a hybrid approach in which two sequential processes, the reader and writer, are clearly identified. The parallelism between the processes in the model is preserved since the code generated for each one is executed as a separate process in the operating system. However, the code of each process is totally sequential. This is not a problem since parallelism exists only between processes. Observe that the main purpose here is to show that it is feasible to synthesize ACM source code from the PN models, and not to introduce an entire new synthesis approach.

The software implementation for ACMs is obtained directly from the Petri net model of each process. The resulting source code is based on the simulation of the net model. In other words, the source code obtained mimics the Petri net behavior. Since the software implementation is intended to execute at some specific environment, it is necessary to guarantee the availability of some basic requirements. More, specifically, the operating system in which the ACM will run should provide the following inter-process communication (IPC) mechanisms:

- Shared memory
- Signal handling

Since the ACM model is built upon the use of a shared memory to communicate the participating processes, the need for the first requirement is obvious. The second requirement is necessary due to the need to provide a process with mechanisms of sending some specific signals to its counter-part. This is particularly necessary for blocking policies in which one of the process blocks waiting for some event. The signal handling must provide support for it.

Briefly, the synthesis method for each process consists of the following steps:

1. Create a shared memory segment which is big enough to hold as many data items as specified by the ACM size;
2. For each place $p$ of the Petri net model, declare a Boolean variable $vp$ named with the label of $p$. If $p \notin EXT$, then initialize $vp$ with the value of the initial marking of $p$;

3. For each transition $t$ of the Petri net model, map $t$ into an `if` statement that is evaluated to true when the values all variables $vp$ corresponding to an input place of $t$ are true. Set the body of the `if` statement properly.

Each process initializes only its own control variables. If $p \in EXT$ then it will in be initialized by the counter-part process, since in this case $vp$ is seen as an external variable. Also, the body of the `if` statement implementing some transition $t$ consists of switching the values of the variables corresponding to the input places of $t$ to **false** and those corresponding to the output places of $t$ to **true**. Besides that, if $t$ models a buffer access action, it is also necessary to add to the body of the `if` the actions needed to write (or read) a new data item to (or from) the communication buffer.

In this work, the ACM source code is synthesized as two C++ classes: one implementing the writer process and the other implementing the reader. Those classes are named **Writer** and **Reader**, respectively. Each ACM process is instantiated as an object of its class.

Algorithms 6.1, 6.2 and 6.3 describe the procedures used to: i) declare the control variables of each process; ii) synthesize the source code of the reader process; and iii) synthesize the source code of the writer process. Of course, the source code obtained depends on the specifics technicalities of the target programming language and the operating system in which the ACM will execute. Besides the fact that those algorithms are generic enough to be used in the generation of code for any operating system that provides the necessary support, all examples discussed in this chapter are specific for the Linux environment. The source code we generate is POSIX (Portable Operating System Interface) compliant, and for this reason we expect it to compile and execute properly at any POSIX machine. However, it has been tested only with the Linux operating system.

## 6.3 Declaring and sharing control variables

In order to perform the steps defined in Section 6.2, 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: i) the declaration of the communication buffer of a given size as a shared memory segment; ii) the declaration and initialization of the control variables; and iii) 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 6.1 defines the procedure for the declaration and initialization of the control variables.

**Algorithm 6.1** Control variables declaration and initialization

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

In the first case, $vp$ is declared as a local Boolean variable that will be shared with the counter-part process and initialized with the value of the initial marking of $p$. In the second case, $vp$ is a shared Boolean variable that was declared and initialized in the counter-part process. In this particular case, it cannot be initialized since it is a read-only control variable, from the point of view of the process being synthesized. Then a reference to a shared variable on the counter-part process is created. Finally, in the third case, $vp$ is declared as a private Boolean variable and is initialized with the value of the initial marking of $p$. In other words, each place is implemented as a single-bit control variable that can be updated only by one process, even if they can be read by both processes.

The method described in this Section has been implemented to generate C++ source code that can be executed properly under Unix environments [4, 60, 72, 73]. More specifically it has been tested in a Linux box with kernel version 2.6.x. For this reason, some details about the implementation should be considered carefully, always taking into account the specific technology being used.

Declaring and initializing regular variables is done in the usual C++ way. For instance, in the writer process it can be found, among others, the following declarations:

```
1 // internal private variables
2 bool wi0;
3 bool pwi0;
```

```
4
5  // internal shared variables
6  bool *we0;
7  int we0_shmid;
8  bool *wne0;
9  int wne0_shmid;
10
11 // external shared variables
12 bool *rne1;
13 int rne1_shmid;
```

Observe that internal variables are declared as simple Boolean attributes of the class, while each shared variable requires two attributes. This is because the way shared memory segments are used in Unix operating systems. Manipulating those segments requires a better comprehension of the Unix mechanisms for interprocess communication (IPC), more specifically, how to use shared memory segments. Observe that, the term *shared memory segment* refers to any piece of memory assigned as shared by the operating system, and not the buffer used to communicate between the processes.

First of all, it is necessary to choose one of the processes to initialize the shared memory segment. In this work it is natural that the segments for each control variable should be initialized by the process that owns the variable. For the communication buffer the writer process has been chosen, but it makes no difference if the chosen one was the reader.

The communication buffer is already declared in the C++ template files as a public attribute of the class that implements the writer process. The type of data to be transmitted has been set to `char`, and to make the schema independent of data type a `#define` statement was used. Depending on the data type to be transmitted it is necessary to change that statement manually. Then the shared memory is allocated and initialized in the constructor of the class by the following code:

```
1  if ((shm_id = shmget(SHMKEY, sizeof(acm_t) * ACM_SIZE, \
2      PERMS | IPC_CREAT)) < 0) {
3
4      cerr << "Writer error! Cannot exec shmget()" << endl;
5      exit(errno);
6  }
7
8  if ((shm_data = (acm_t *) shmat(shm_id, (acm_t *) 0, 0)) == \
9      (acm_t *) -1) {
10
11     cerr << "Reader error! Cannot exec shmat()" << endl;
12     exit(errno);
13 }
14
```

```
15 *shm_data = '-';
```

On the above, `shm_id` represents the identifier given by the kernel of the operating system to the shared memory segment that is created through the `shmget()` system call in line 1. In particular, the second parameter of the system call specifies the size of the segment to be allocated, which in this case is defined as the size of one data item to be transmitted (`acm_t` that here is a `char`) multiplied by the size of the ACM. Also, note the flags `PERMS | IPC_CREAT` which specify that a new shared segment should be created.

Then in line 8, `shmat()` is used to attach the new shared memory segment identified by `shm_id` to the address space of the calling process, i.e. to the attribute `shm_data`. Finally, in the last line, the buffer is initialized with a simple dash.

The source code above was used to implement the communication buffer used in data transfers between the processes. A similar schema is used to share all control variables of the system, with the important difference that in the process that only reads a certain variable, the flags on the `shmget()` system call do not specify the creation of a new shared segment.

## 6.4 Synthesizing the data access and control

Up to now it has been discussed how to declare the control variables, but the control part itself has not been synthesized, and there is no indication on how the data is passed from one side to the other. The synthesis of the control for the reader and writer processes is introduced by Algorithms 6.2 and 6.3, respectively.

In Algorithm 6.2 the synthesis method for the control of the reader process is introduced. The first case captures the synthesis of control of a data access transition. More specifically, it is a *read transition* addressing the $i^{th}$ cell. Remember that in the notation introduced in Section 4.2.1, $(t, i) \in M_a$ denotes a transition $t$ that accesses the $i^{th}$ data cell. The control condition is given by the pre-set of $t$, and if it is satisfied then all variables corresponding to a place in its pre-set are switched to `false` and the variables on its post-set to `true`. Finally, some data is read from the $i^{th}$ cell. The second case captures the synthesis of control of a *control transition*. As previously, the condition is given by the pre-set of $t$, and then the variables on the pre-set is switched to `false`, and the variables on the post-set to `true`. Note that in the last case there is no need to read data from the buffer.

Algorithm 6.3 is similar to Algorithm 6.2. Since it refers to the synthesis of source code of the writer process, in line 7 a data write operation instead

**Algorithm 6.2** 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 $vp = true$
5: $\forall p \in \bullet t$ add to the **if** body $vp := false$
6: $\forall p \in t\bullet$ add to the **if** body $vp := true$
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 $vp = true$
11: $\forall p \in \bullet t$ add to the **if** body $vp := false$
12: $\forall p \in t\bullet$ add to the **if** body $vp := true$
13: **end if**
14: **end for**

of data read one is added. More specifically, the process will address the $i^{th}$ cell.

### 6.4.1 Atomic transitions

In Algorithm 6.2 it can be noticed that transitions of the Petri net model, which are atomic, are mapped into non-atomic actions. The non-atomicity of the **if** statements in the C++ code may introduce undesired behavior during runtime. However it is very unlikely that this bad behavior happens. The critical situation occurs when both processes are about to point to the same cell/slot. Let us analyze the problem for each policy. For re-reading only ACMs, which have only one slot per cell, there are two critical moments:

1. When the buffer is full of non-read items and the writer attempts to write a new one;

2. When the buffer if empty of non-read items and the reader attempts to read a new one.

In the first case, the writer will try to advance to the next cell, which the reader is pointing to, and then it will block until the cell is released. If the writer tests the next cell before the reader finishes releasing it, the writer will stay where it is. Otherwise, it will advance. It is necessary that the reader blocks its own next cell before releasing the current one. In this

**Algorithm 6.3** Synthesis of control for the writer

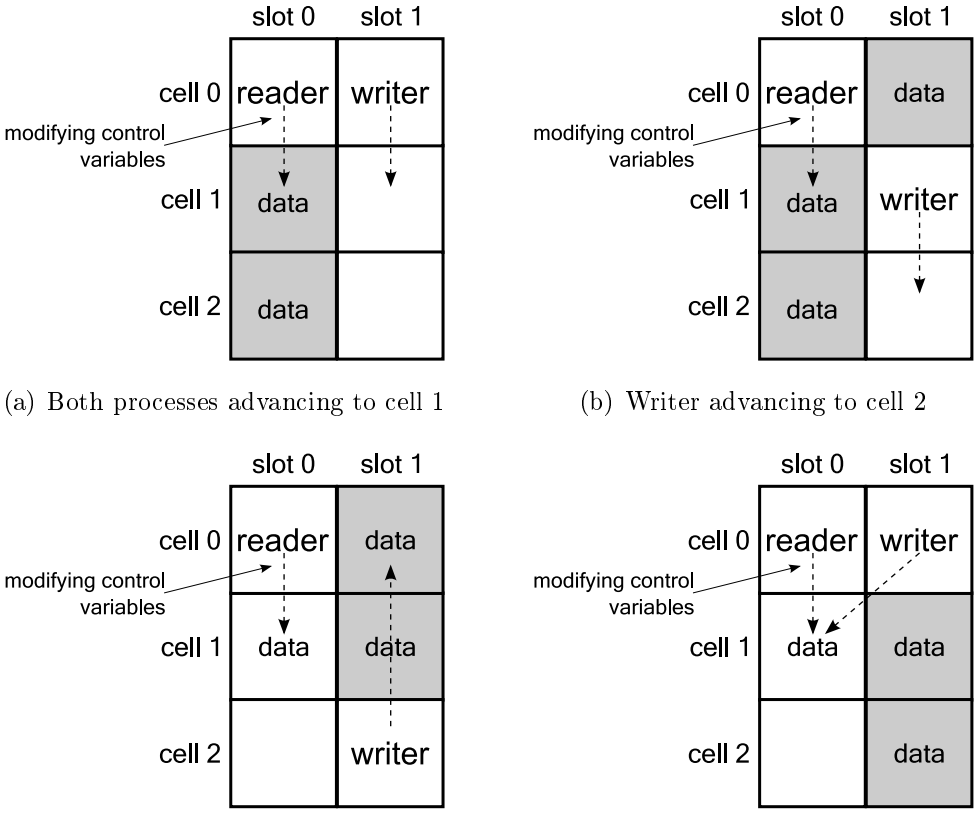
1: **for all** $t \in T$ **do**
2: **if** $t \in T_a$ with $(t, i) \in M_a$ **then**
3: Create new **if** statement
4: $\forall p \in {\bullet}t$ add to the **if** condition $vp = true$
5: $\forall p \in {\bullet}t$ add to the **if** body $vp := false$
6: $\forall p \in t{\bullet}$ add to the **if** body $vp := true$
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 $vp = true$
11: $\forall p \in {\bullet}t$ add to the **if** body $vp := false$
12: $\forall p \in t{\bullet}$ add to the **if** body $vp := true$
13: **end if**
14: **end for**

case, the reader will point to two cells at the same time before finishing its operation. The writer will only be allowed to advance after the reader releases its current cell, and in this case the new next cell is already locked by the reader. Proceeding in this way, the bad behavior will not occur. The same reasoning applies to the second case.

For overwriting ACMs, the problem is more complicated. As stated before, the writer always attempts to write on different slots every time it passes through each cell. When both processes are accessing the same cell (different slots) and if both advances at the same time they will also point to different slots in the same cell. The reader will point to the slot containing the fresh data item, which was accessed by the writer on the previous cycle. While the writer will point to the slot that not been accessed in the previous cycle.

The problem occurs when the writer is faster than the reader. At some moment it will advances through the circular buffer and reaches the cell the reader is point to again. At this moment, both processes will prepare to advance to the same cell/slot. If this particular situation happens, the communication protocol will present an undesired behavior. However, it is interesting to notice that it only happens if the writer is able to generate data items to fulfill the buffer, and store them there, even before the reader is able to read only one data item. This is very unlikely to happen. To better understand the problem let us analyze the following situation. A 3-cell OWRRBB ACM that is full of non-read data items, both processes are pointing at the same cell (different slots), and both prepare to advance to

the next cell at the same time. They will first test the values of the control variables and, if the slot is free, they will change the values of the variables in order to advance. This situation is illustrated in Figure 6.1(a).




(a) Both processes advancing to cell 1

(b) Writer advancing to cell 2

(c) Writer advancing to cell 0

(d) Writer advancing to cell 1

Figure 6.1: Execution of OWRRBB ACM with 3 cells.

If, for some reason, the reader blocks while advancing but after testing the control variables, it may happens that the writer advances to cell 1, then to cell 2, and then back to cell 0 before the reader finishes its operation. In this case, the next slot the writer will access is the same the reader is advancing to, as illustrated in Figure 6.1(d). This is because the writer always advances to the slot holding the oldest data item in the cell, and also because the reader is blocked and has not release its current slot or locked the next one. In this very specific case, the implementation fails. However, note that it may be very unlike that the writer will be able run over the entire buffer without the reader being able to set few Boolean variables.

One can argue that if the reader looses the CPU for a long time this situation may occurs frequently, especially for data items of small size. To guarantee that the implementation does not fail, it is necessary to ensure that the process cannot lose the CPU when advancing to the next cell. In other words, it is necessary to ensure that the statements implementing the behavior of a control transition should be executed in the same CPU slice the process has allocated at a given time. Note that this is not the same of an atomic action.

The discussion above can bring the perception that the asynchrony between the processes is affected. This is partially true. The ACMs are mainly designed to communicate processes running on different time domains, such as different CPUs with their own clocks. In this way, giving a process the warranty it will not lose the CPU when performing an action does not affects the other process. Also, it is not the same as protecting the access to a critical region with semaphores, since the other process is always allowed to proceed if the control variables schema is respected.

### 6.4.2 Synthesis of the data access actions

These procedures have been used to generate C++ code that implements the behavior for the ACMs. As previously, implementation issues related to the specific operating system being used should be considered. For instance, the synthesized data access actions, for both writer and reader processes of a 3-cell re-reading ACM, are given by the C++ source code below. First let us look at the writer.

```
1  void Writer::Send(acm_t val) {
2
3      if (w0 == true) {              //wr0
4
5          w0 = false;
6          pw0 = true;
7          *(shm_data + 0) = val;
8      } else if (w1 == true) {       //wr1
9
10         w1 = false;
11         pw1 = true;
12         *(shm_data + 1) = val;
13     } else if (w2 == true) {       //wr2
14
15         w2 = false;
16         pw2 = true;
17         *(shm_data + 2) = val;
18     }
19 }
```

In the above it is possible to see the `Send()` method, which actually writes some data into the ACM. It belongs to the **Writer** class. It receives a data item of type `acm_t` to be transmitted, and should be accessed through an object of the class. Line 3 implements the test of the pre-set of transition $wr_0$ in the Petri net model. If such transition is enabled in the model, then the variables corresponding to its input places should have the value `true`. In this case, the variables implementing the pre-set of $wr_0$ are set to `false`, which is implemented in line 5. Then, the variables implementing the post-set of $wr_0$ are set to `true` in line 6. Then, some data is written into the $0^{th}$ cell of the ACM, which is implemented in line 7. Note that the parameter `val` is the new data item to be sent, and `shm_data` implements the communication buffer. The same reasoning applies to the other `if` statements in lines 8 and 13, in which the test of the pre-sets of transitions $wr_1$ and $wr_2$ are implemented. Finally, it should be observed that the accesses to the correct data cell in the buffer are determined by simple pointer arithmetic. For instance, to access the second position in the communication buffer, `*(shm_data +1)` is used in the source code. Then, the `Receive()` method of the **Reader** class is introduced.

```
1  acm_t Reader::Receive(void) {
2
3      acm_t val;
4
5      if (r0 == true) {          // rd0
6
7          r0 = false;
8          pr0 = true;
9          val = *(shm_data + 0);
10     } else if (r1 == true) { // rd1
11
12         r1 = false;
13         pr1 = true;
14         val = *(shm_data + 1);
15     } else if (r2 == true) { // rd2
16
17         r2 = false;
18         pr2 = true;
19         val = *(shm_data + 2);
20     }
21
22     return(val);
23 }
```

The same reasoning applies to the reader access method shown above. The only difference is that instead of writing into the buffer, it reads some data from there and returns it to the calling method. Note that in both

cases, the data access action does not require the corresponding process to wait for any reason. This is because when a process is ready to access a certain cell, the control actions had already been taken, and the correct cell to be addressed has been determined previously.

### 6.4.3 Synthesis of the control actions

The methods implementing the control actions are somewhat more complex due to the fact that it is necessary to provide a block mechanism to some of the processes depending on the ACM policy. However, they follow the same principle. For instance, applying Algorithm 6.3 the implementation of the writer's control actions, i.e. the $\lambda$ actions, would generate the C++ source code below.

```
1  void Writer::Lambda(void) {
2
3      while (true) {
4
5          if (*we0 == true && *wne1 == true &&
6              w0p == true && *rne1 == true) { // l0_1
7
8              // set pre-set to false
9              *we0 = false;
10             *wne1 = false;
11             w0p = false;
12
13             // set post-set to true
14             w1 = true;
15             *wne0 = true;
16             *we1 = true;
17
18             // leave the loop
19             break;
20         }
21
22         if ( . . . ) { // l1_2
23
24             . . .
25         }
26
27         if ( . . . ) { // l2_0
28
29             . . .
30         }
31
32         pause();
33     }
```

```
34 }
```

As before, each transition is implemented as an `if` statement whose condition is given by the variables of the pre-set of the transition and the body consists of switching the variables of the pre-set to `false` and the variables of the post-set to `true`. For example, the code implementing the firing of transition $\lambda_{01}$ is given by lines 5 to 20 in the piece of C++ code above. Note that `we0` and `wne0` stands for $w = 0$ and $w \neq 0$, respectively.

It is important to observe that the control actions of the writer process are inside an infinite loop whose last instruction is a call to the `pause()`[1] library function. This is done because if the reader is pointing to the next cell, then there will not be any $\lambda$ transition enabled. With the writer pointing to the $i^{th}$ cell, it means that the reader is pointing at the $(i+1)^{th}$ cell. In this case the writer should wait for the reader to execute, and using the `pause()` function is the way of doing it on Unix. This also avoids busy waiting. The `pause()` is terminated when the reader advances to its own next cell and sends a signal to the writer indicating it. Then the writer executes again and tries to advance to its next cell. The infinite loop is broken by a `break` statement in line 19.

The control actions of the reader process are implemented by the `Mu()` method. Again, each transition is implemented as an `if` statement. For instance, $\mu_{00}$ is implement by the code from lines 3 to 7 and $\mu_{01}$ is implemented in lines 8 to 17. Observe that every time the reader executes a control actions, it sends the `SIGCONT` signal to the writer, as in lines 7 and 17. This is to wake up the writer in the case it is sleeping due to a `pause()`. Since this behavior mimics the Petri net model, both processes will not run into a deadlock due to some race condition.

```
1  void Reader::Mu(void) {
2
3      if (r0p == true && *we1 == true) { // m0_0
4
5          r0p = false;
6          r0 = true;
7          kill(pair_pid, SIGCONT);
8      } else if (*re0 == true && *rne1 == true &&
9                  r0p == true && *wne1 == true) { // m0_1
10
11         *re0 = false;
12         *rne1 = false;
13         r0p = false;
14         r1 = true;
```

[1] 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.

```
15          *rne0 = true;
16          *re1 = true;
17          kill(pair_pid, SIGCONT);
18      } else if ( . . . ) { // m1_1
19
20          . . .
21      } else if ( . . . ) { // m1_2
22
23          . . .
24      } else if ( . . . ) { // m2_2
25
26          . . .
27      } else if ( . . . ) { // m2_0
28
29          . . .
30      }
31  }
```

### 6.4.4 Processes flow

The methods generated above need to be integrated into the communicating processes. As explained before and shown in Figures 6.2 and 6.3, 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 objects, respectively. With this, the correct use of the communication scheme is ensured.

In this chapter an automatic approach to generate source code from Petri net models was discussed. The algorithms introduced here only give conceptual 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 operating system. The reader should consult [72] and [73] for more details on creating shared memory segments on Unix like and POSIX operating systems.

The generation of C++ code for the overwriting policies is done by executing Algorithm 6.1 and some variants of Algorithms 6.3 and 6.2. These variants differs from the original algorithms in the fact that they use $M_a$ and $M_c$ as presented in Definition 4.8 instead of Definition 4.3. It is also necessary to point out that since in overwriting policies there are two slots per cell, a different scheme to implement the communication buffer is used. But it still uses simple C++ pointer arithmetic.

More specifically, for an overwriting ACM of size $n$, it allocated enough

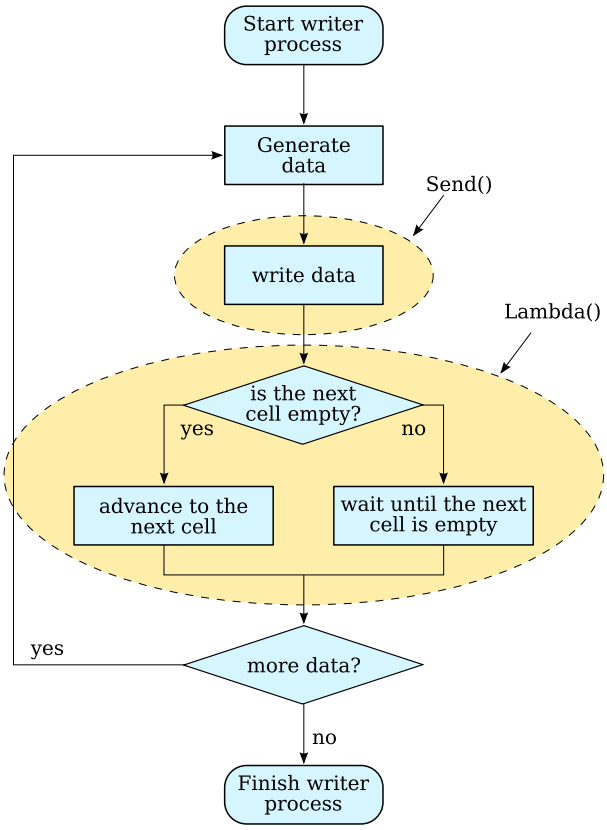



Figure 6.2: Flowchart for the writer process

space for $2 \times n$ data items. The available space is grouped into pairs that will represent the pair *(cell,slot)*. For instance, the cell 00 corresponds to `*(shm_data + 0)`, 01 corresponds to `*(shm_data + 1)`, 10 corresponds to `*(shm_data + 2)` and so on. The offset needed to address the pair (cell,slot) $ij$ is given by $i * 2 + j$.

In the source code a read operation will appear as

```
val = *(shm_data + 0);
```

while a write operation appears as

```
*(shm_data + 0) = val;
```

The complete source code for the 3-cell RRBB example used to illustrate the C++ code synthesis presented in this chapter is included in Appendix B.1. Additionally, links to complete examples of C++ code for 2 cells OWBB and OWRRBB ACMs are also included.

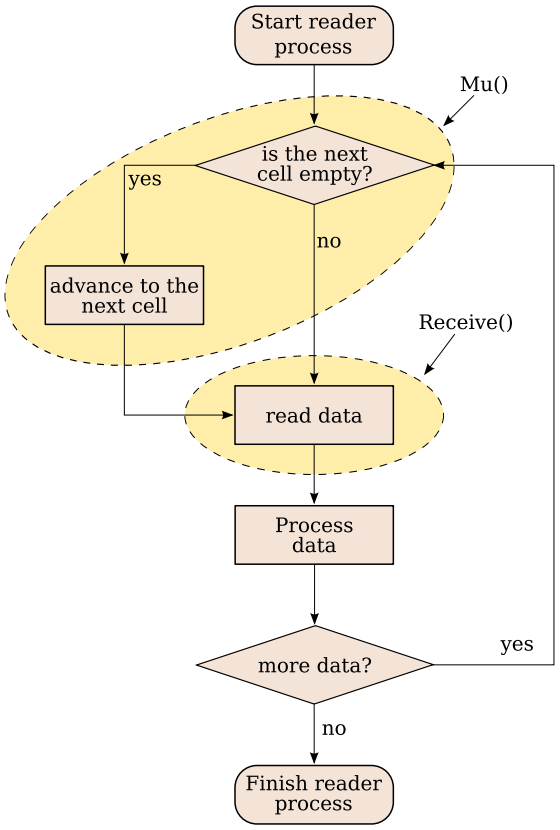



Figure 6.3: Flowchart for the reader process

## 6.5 Conclusions

In this chapter the automatic generation of C++ source code for the ACMs was introduced. The algorithms used in the synthesis of the ACM were detailed and the operating system requirements for the code to work correctly were outlined. The synthesis of ACMs has been explained using the generation of C++ source code for the particular case of the Linux operating system.

The source code is synthesized in such a way that it mimics the behavior of the Petri net model of the ACM. For this reason and since the Petri net satisfy coherence and freshness, the code is expected to satisfy these properties.

The synthesis procedure outlined in this chapter has been used to generate a number of ACM implementations of different sizes. The source code was compiled, and a set of tests were performed in order to check that the program behave as expected. As far as the tests show, no anomalies have

been observed.

The reader may have observed that the generated C++ code is very regular (because the original Petri nets are regular). One could develop a single piece of code (per ACM policy) that would take the size N of the ACM as a parameter, rather than generate separate pieces of code for each individual N. However, proceeding like that, the designer will lose the ability of generating code for different protocols. As the methodology is defined, the designer needs only to define the PN modules following the definitions presented in this, and the source code can be obtained. Besides that, since we will be dealing with n-ary variables, we cannot even guarantee that if both processes refer to the same variable at the same time, the reader process will recover a valid value. For instance, a 3-cell ACM will require a two bits variable to count the cells, and this 3-bit variable can represent four values. This may be a serious problem.

Another issue with the C++ code generated is related to performance, since the wide use of signals to communicate between the processes may severely degrade the performance of the system. First of all, consider that the main concern in this work was to demonstrate that the methodology was feasible. Signals are only a particular way of implementing the solution for a particular operating systems. Performance is a real problem that need to be addressed, and more study is necessary to propose an implementation that minimizes (or even does not use) signals. One possibility is to use DBUS[2] to manage process communication.

[2] *"D-Bus is a message bus system, a simple way for applications to talk to one another. In addition to interprocess communication, D-Bus helps coordinate process lifecycle; it makes it simple and reliable to code a "single instance" application or daemon, and to launch applications and daemons on demand when their services are needed."* [2].

# Chapter 7

# Automatic synthesis of ACM hardware implementations

In this chapter the synthesis of hardware implementation of ACMs will be introduced. Firstly, a generic design will be presented as a set of block diagrams. This design can be used to obtain ACMs for both re-reading and overwriting policies.

Due to this fact, the block diagrams are not complete in the sense that the control of the reader and writer processes is not explicitly described there. For this purpose, a set of Finite State Machines (FSM) describing both processes are introduced for the RRBB policy. Since the FSM is specific for an ACM of a certain size, a procedure to obtain the FSM for ACMs of any given size is provided. Then it is discussed how the FSMs are related to the Petri net models introduced on Chapter 4. Finally, the generation of Verilog source code from the FSMs will be introduced.

The generation of hardware descriptions for the overwriting policies will not be addressed here, only the re-reading one. However, the same schema can be extended to support the generation of overwriting ACMs. For this it is necessary only to define the proper FSM, and substitute it into the design flow.

## 7.1 Block diagrams design

The general structure for an ACM hardware implementation is introduced by the block diagram in Figure 7.1. Three main entities are defined. Two of them represent the reader and writer applications that are provided by the user of the ACM. The design of such applications is covered here. The third entity is the ACM itself, which is the subject of the present work.

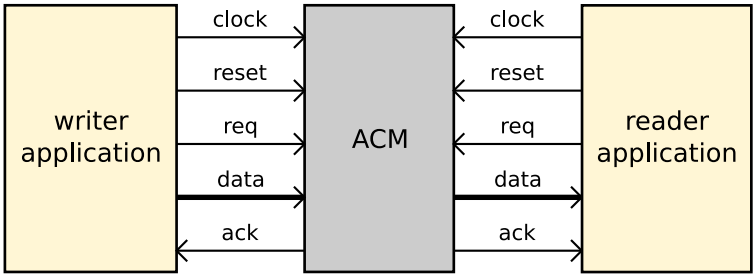



Figure 7.1: The ACM general structure

As can be seen in Figure 7.1, the ACM connects to both, reader and writer applications. Observe that the ACM receives the **clock** signal from both writer and reader, meaning that part of it will execute in the same clock domain as the reader and part will execute in the clock domain of the writer application. The ACM also receives requests from both trough the **req** signals and returns acknowledgments when the task has been performed through **ack**. Finally, the ACM receives some data from the writer and sends some data to the reader using the **data** buses.

Despite having the same labels in Figure 7.1, the signals connecting the ACM to the writer application are not physically the same as those connecting the ACM to the reader application. As stated above, the focus of this chapter is on the design of the ACM, and not on the applications that use it. For this reason, from now only the design of the ACM will be addressed.

The block diagram of the ACM is shown in Figure 7.2. The ACM is composed of three modules: i) the writer module; ii) the reader module; and iii) the shared memory module. Clearly these reader and writer modules are not the reader and writer applications of Figure 7.1. The writer and reader modules belong to different clock domains. Each of them has its own clock and reset signals, and they receives requests from the external applications. On the other hand, the shared memory module does not share the clock with either the writer or the reader modules, but it can react to stimuli from both modules in order to store or recover some data item. Besides that, the writer and reader modules are connected to allow each module to test the control variables of its counter-part module.

Each internal module of the ACM communicates with the other modules using the proper internal signals. For instance, when the writer module receives a request from the environment, which is indicated by **w_req=1**, it first makes a request to the shared memory module setting **wreq** to 1 and then it waits for the **wack** signal. After receiving **wack**, it forwards the signal to the writer application using the wire **w_ack**. Then it checks if the

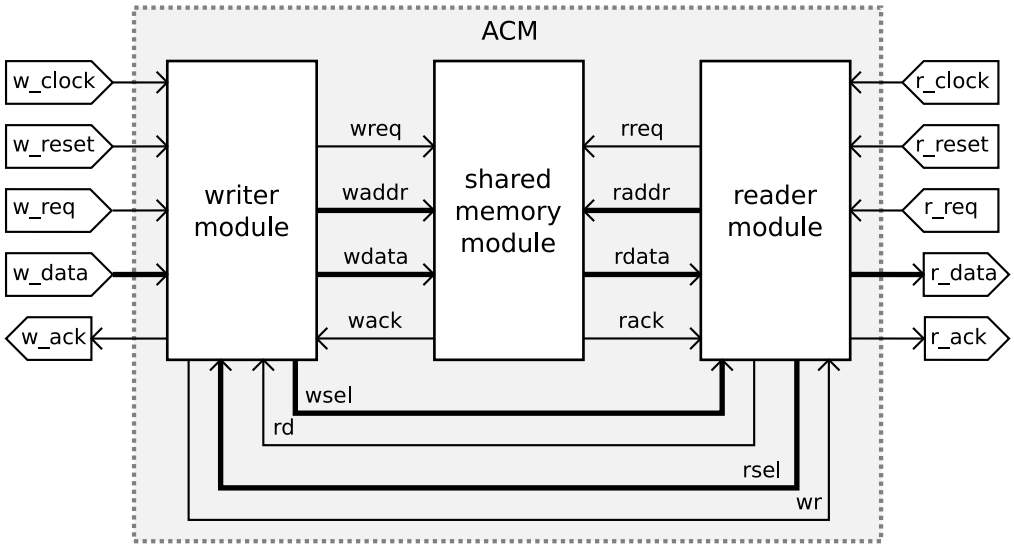



Figure 7.2: The ACM block diagram

reader module is accessing the next cell by setting **wsel** to the value of the next cell and checking the value of the input signal **rd**. If the reader is not pointing at the next cell, the new data is released for reading and the writer module prepares to write the next cell. Otherwise it waits until the reader is not pointing to the next cell any more, if the ACM in question implements a re-reading policy, or it discards some data item if the ACM implements an overwriting policy. The behavior of the reader is similar to the behavior of the writer, except the fact that if the writer module is pointing to the next cell, the reader prepares to re-read the current cell.

The writer module is detailed in the block diagram of Figure 7.3. The control variables of the writer are implemented by the flip-flops named **w0**, **w1**, $\cdots$, **wn** on the left-bottom side of the diagram. Each flip-flop implements one control variable, and they are one-hot encoded, meaning that exactly one of them must have the value set to 1 at any time. At each tick of the clock, the writer engine updates the values of the control variables if necessary.

The reader module, which is shown in the block diagram of Figure 7.4, is similar to the writer module. The main difference is that it receives data from the shared memory module and returns this data to the reader application. Observe that in both cases, a module *"asks"* the other if a cell is being accessed or not by setting the value of its select signal properly, and the answer comes by the corresponding result signal (**rd** or **wr**). In any case, the result is only perceived after passing through two sequential flip-flops clock signals, which requires two clock signals. This is necessary in order to minimize the

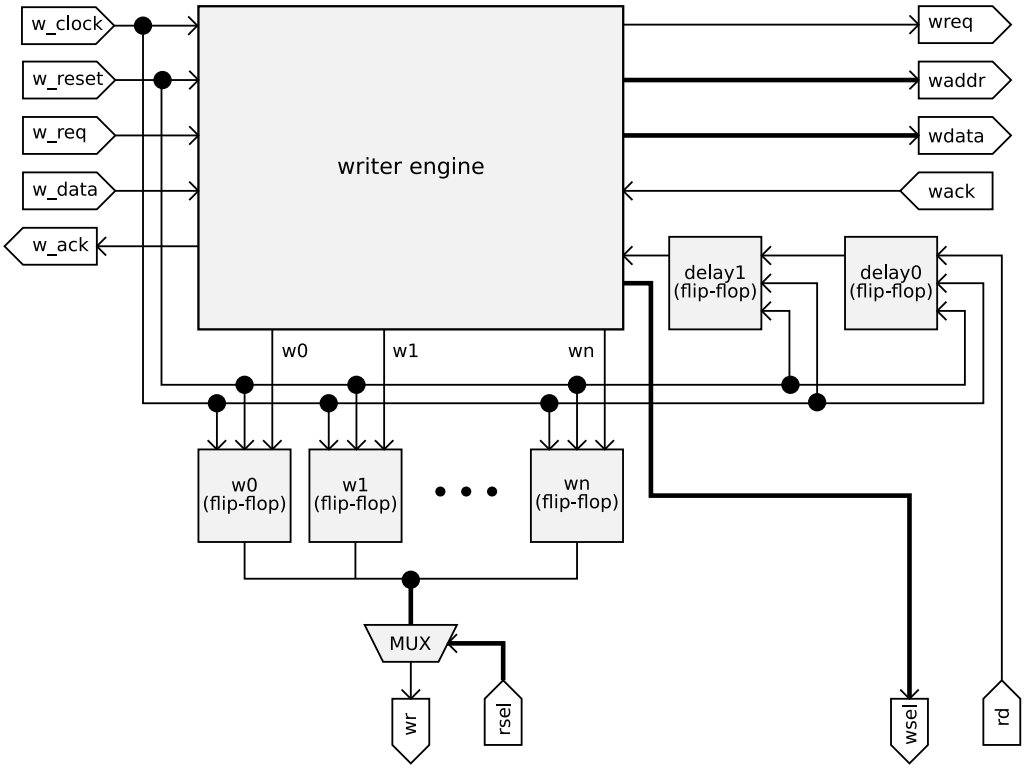



Figure 7.3: The writer block diagram

probability of metastability problems since the access to the control variables is not controlled by any mutual exclusion mechanism.

Observe that such block diagrams do not specify any specific behavior about the writer or the reader, neither how the control variables are updated. This is specified in the writer/reader engine sub-modules. Using a schema similar to the one used in Section 6.4 to encode the pair $(cell, slot)$ of the overwriting policies, it is possible to use the same block diagram of Figures 7.3 and 7.4 to design ACMs for both policies. The only need is to replace the writer engine properly. Of course, the number of flip-flops needed to design ACMs of different sizes changes according to the amount of cells needed by the ACM.

## 7.2 The finite state machines of the engines

The last step needed to complete the design of a hardware implementation of ACMs is to specify the behavior of the writer and reader engines. In this work the behavior of each engine was defined as a Finite State Machine

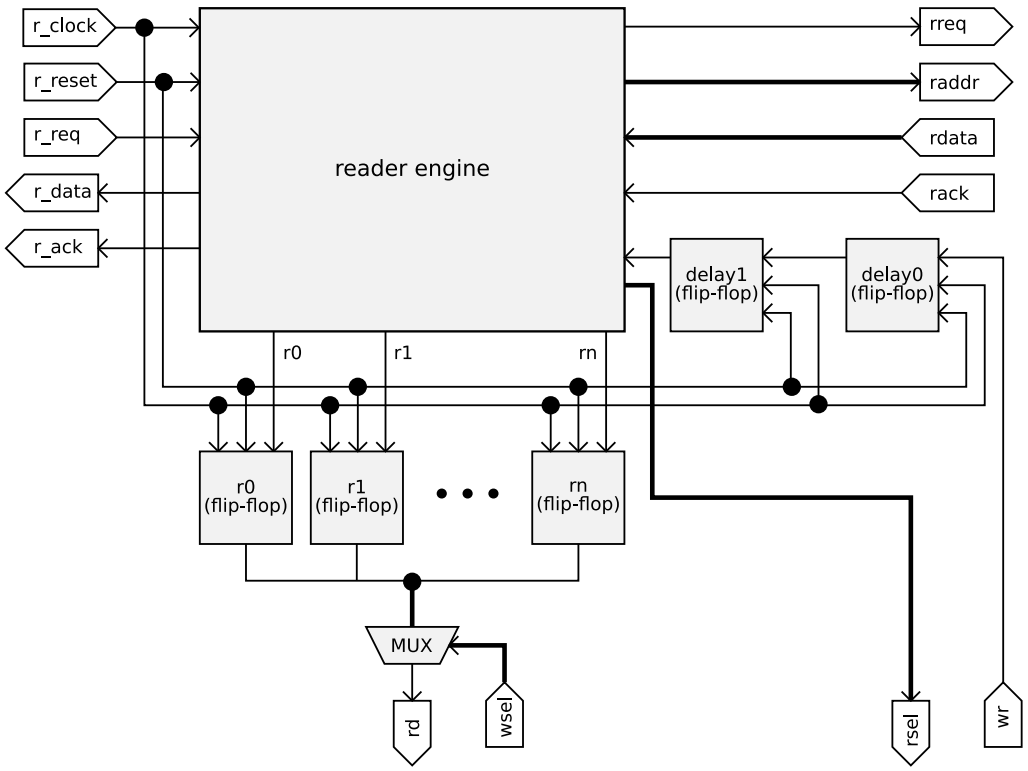



Figure 7.4: The reader block diagram

(FSM). Since there are ACMs of different sizes, there will be one FSM for each possible ACM size. In this section it will be described how to generate FSM specifications for RRBB ACMs. For this purpose, the FSM for the smallest possible RRBB ACM will be introduced first. Then it will be argued how to obtain ACMs of any size from it.

In Figures 7.5 and 7.6 the FSMs of the writer and reader engines for a 3 cells RRBB ACM are shown. Initially the writer is in an idle state ready to access the cell number 1, as indicated in the state labeled **idle1**. The writer is already pointing to cell 1, and the next cell has also been selected, indicated by **wsel=2**. When a writing request is received, the state changes to **init1**, and a request is made to the shared memory module. These action are indicated in the FSM by **w_req=1** and **wreq=1**, respectively.

Once the request has been processed by the shared memory module, the **wack** signal is received and the state changes to **end1**. Then the writer engine checks if the reader is accessing the next cell by testing **!rd**. Remember that **wsel** has already been set at state **idle1**. If the reader is not accessing that cell, then the writer advances to it updating its own control variables,

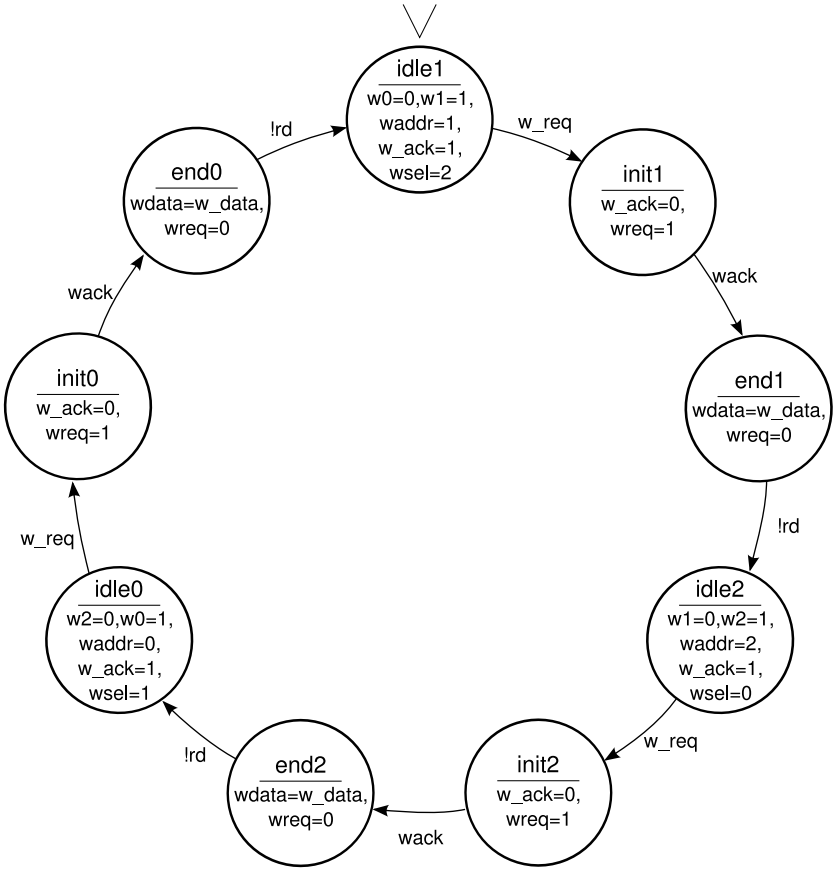



Figure 7.5: The writer finite state machine

sends an acknowledgment signal to the application served and checks one more cell ahead of the reader. This is indicated in the FSM by **waddr=2**, **w1=0**, **w2=1**, **w_ack=1** and **wsel=0**, respectively. This cycle is repeated until the writer returns to the state **idle1**.

The reader FSM in Figure 7.6 behaves much in the same way as the writer FSM. The main difference compared with the writer FSM in Figure 7.5 is that in order to finish a data access action the reader does not block (while the writer blocks if the reader is pointing to the next cell). Instead, it returns to the state in which it is ready to access the cell it has just read. In other words, it prepares to re-read the current cell, and the values of the control variables are not modified. Note the arcs with conditions **!wr** and **wr** starting from the states labeled **end0**, **end1**, and **end2**. These arcs indicate that the status of the writer does not block the reader. The mechanism to access the control variable of the writer module is the same, and it communicates with the shared memory module in the same way as the writer.

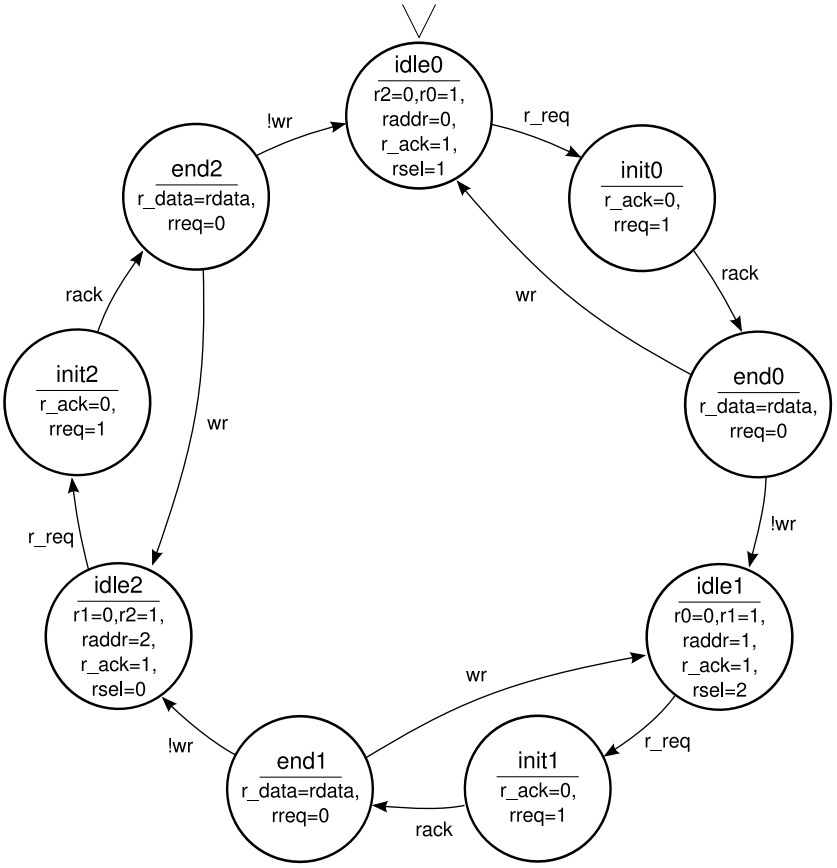



Figure 7.6: The reader finite state machine

Finally, in Figure 7.7 the FSM for the shared memory module is shown. This is the module actually responsible for executing the data access operations in the communication buffer, while the reader and writer modules control where and when these operations are done. The shared memory module communicates with both writer and reader modules and reacts to their stimuli. It receives a request, a data to be stored and the address to store the data from the writer module. After saving the data it returns an acknowledgment signal indicating the termination of the action. It also receives a request and an address from the reader, and returns the data requested and an acknowledgment signal. Observe that the write and read operations can be performed concurrently.

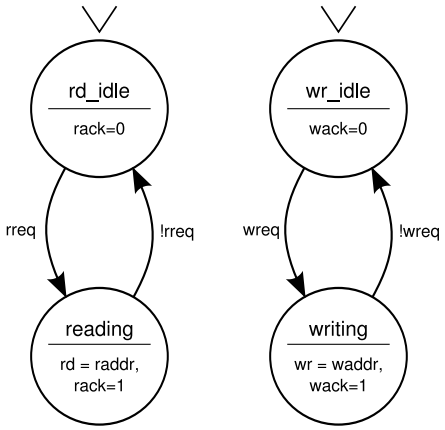



Figure 7.7: The shared memory finite state machine

## 7.3 Automatic generation of FSM specifications

From Figures 7.5 and 7.6 it is easy to observe that each FSM has a basic pattern that is replicated in larger models. This happens due to the fact that the control over each ACM cell is exactly the same, differing only in the addresses they control the access to. For instance, it can be observed that the writer FSM can be easily obtained from the FSM module shown in Figure 7.8.

More specifically, in order to obtain the FSM of the 3-cell RRBB ACM described previously, it is only needed to instantiate a number of FSM modules like the one in Figure 7.8 and connect them properly. Since this FSM module expresses all the control needed for one ACM cell, the total number of modules needed corresponds to the size of the ACM, e.g. for the 3-cell RRBB above, three FSM modules are needed. To instantiate the FSM module for the $j^{th}$ cell it is necessary to generate an FSM of the module and replace all occurrences of the strings **I**, **J** and **K** properly (**I**, **J** and **K** represent the number of the previous, the current and the next cell respectively, and they must be replaced by $j-1$, $j$ and $j+1$, respectively).

Finally, it is necessary to connect the obtained FSM modules. This is easily done by just merging the output arc labeled **!rd** of the $j^{th}$ FSM module with the input arc labeled **!rd** of the $(j+i)^{th}$ FSM module. Obverse that we are considering the operation $(j+1)$ as $((j+1) \mod n)$, where $n$ is the ACM size. After these two simple steps the FSM of the writer engine of an RRBB ACM is obtained. The synthesis of the FSM for the writer engine is described in Algorithm 7.1.

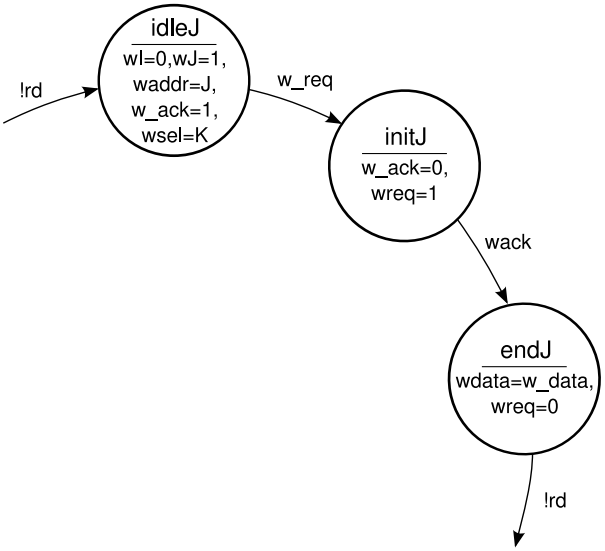



Figure 7.8: FSM writer module

The same procedure is applied to obtain the FSM for the reader engine. The FSM module for the reader is shown in Figure 7.9.

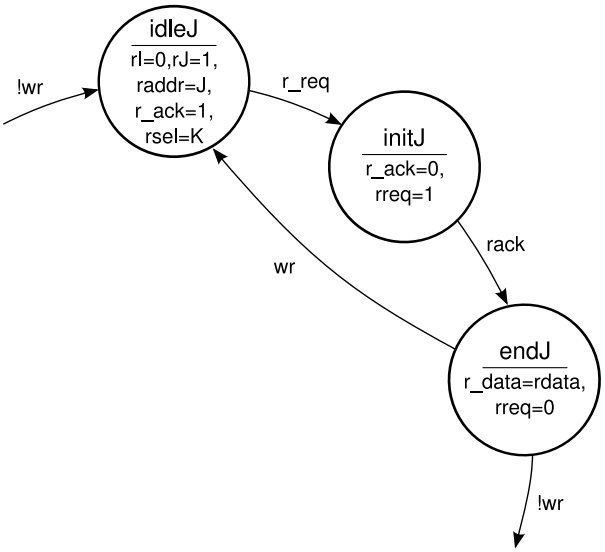



Figure 7.9: FSM reader module

The attentive reader may have noted that the FSM models described above do not corresponds to the Petri net models introduced in Section 4.3. This is because in the Petri net models presented in Chapter 4 all data access operations are treated as atomic, while in the FSM models above they are

**Algorithm 7.1** Synthesis of the FSM for the writer engine

1: **for** $j = 0$ to $n - 1$ **do**
2: Create new writer module as in Figure 7.5
3: Replace I by the value of $j - 1$
4: Replace J by the value of $j$
5: Replace K by the value of $j + 1$
6: **if** $j \neq 0$ **then**
7: Merge input arc of the $j^{th}$ module with the output arc of the $j - 1^{th}$ module
8: **end if**
9: **if** $j = (n - 1)$ **then**
10: Merge output arc of the $j^{th}$ module with the input arc of the $0^{th}$ module
11: **end if**
12: **end for**

not. This may lead one to think that coherence and freshness properties may not be preserved on the implementations obtained from those FSM models.

In order to guarantee those properties it is necessary to modify the Petri nets to model the data access operations as non-atomic actions. This would require to split each transition $t$ modeling the data access action into two new transitions, $t_i$ and $t_e$. $t_i$ models the initiation of a data access action, while $t_e$ models the end of the data access action. The input places of $t$ should be input places of $t_i$, and the output places of $t$ should be output places of $t_e$. Finally, $t_i$ and $t_e$ should be connected by a new place. Then, the PN models should be regenerated and subjected to model checking again. Clearly, this approach increases the number of variables of the system, which also affects negatively model checking performance.

From the Petri net modules used to build the models for RRBB ACMs it is easy to observe that each transition that will be split has exactly one input and one output place. This true for both writer and reader processes, as can be seen in Figures 4.2(a) and 4.2(b), respectively. This means that the modifications needed in the PN model will not change the sequence that the cells on the ACM are addressed by the processes. This is necessarily true because the two new transitions refer to the same data access action as the old one. Consequently, the freshness property is also satisfied in the new Petri net model.

In order to argue why coherence is also satisfied in the new Petri net model, the alternative coherence verification discussed on Section 5.4 will be of great value. There, it has been proposed that coherence is also satisfied

if for any cell $j$ in the ACM, the total amount of tokens on the input places and output places of the data access transitions of $j$ does not exceed one. In other words, for all cell $j$, $M(w_j) + M(pw_j) + M(r_j) + M(pr_j) \leq 1$. This has been verified before, and has been proved true for a number of ACMs instances.

In the new Petri net model, this way of describing coherence is mapped to $M(w_j) + M(pw_j) + M(nw_j) + M(r_j) + M(pr_j) + M(nr_j) \leq 1$, where $nw_j$ and $nr_j$ correspond to the new places added when splitting transitions $wr_j$ and $rd_j$ respectively. It is obvious that this new constraint is preserved in the new Petri net model, and consequently coherence is also satisfied.

## 7.4 Verilog code synthesis

To complete the hardware synthesis for RRBB ACMs introduced in this chapter, it is necessary to point out how to obtain an artifact that can be synthesized as a physical hardware, from the set of FSMs presented in the Sections above. In this Section it will be outlined how to obtain a Verilog [77] code for RRBB ACMs.

The block diagrams described in Section 7.1 are used to generate a set of templates in the Verilog language for ACMs of any size. It is only necessary to take care to setup correctly the data type to be transmitted and the size of the ACM. Note that in the ACM module this size does not appear explicitly, however the signals **wsel** and **rsel** depend on it. More specifically, these signals should have $log_2 n - 1$ wires, where $n$ is the size of the ACM. Besides that, it is also necessary to instantiate a number of flip-flops corresponding to the size of the ACM in both reader and the writer Verilog modules. As explained in Section 7.1, each flip-flop corresponds to a binary variable that controls the access to a specific cell. Also, all wires should be correctly connected. Since the Verilog template files are almost static, their setup will not be detailed here.

The main problem is in the synthesis of the code that controls the access to the shared memory, i.e. in the synthesis of the code corresponding to the writer and reader engine sub-modules of Figures 7.3 and 7.4. These sub-modules are described by the FSMs introduced in Section 7.2. The Verilog code generation uses the simple idea of getting each FSM module used to build the writer engine or the reader engine and mapping it to a piece of Verilog code that is equivalent to the FSM module. The FSM module of the writer engine is mapped to the following piece of Verilog code:

```
1 state[IDLE_J]:
2 begin
```

```
3       if (w_req) begin
4           w_ack <= 1'b0;
5           wreq <= 1'b1;
6           state[IDLE_J] <= 1'b0;
7           state[INIT_J] <= 1'b1;
8       end
9   end
10  state[INIT_J]:
11  begin
12      if (wack) begin
13          wreq <= 1'b0;
14          wdata <= w_data;
15          state[INIT_J] <= 1'b0;
16          state[END_J] <= 1'b1;
17      end
18  end
19  state[END_J]:
20  begin
21      if (!rd) begin
22          w_J <= 1'b0;
23          w_J+1 <= 1'b1;
24          wsel <= J+2;
25          w_ack <= 1'b1;
26          waddr <= J+1;
27          state[END_J] <= 1'b0;
28          state[IDLE_J+1] <= 1'b1;
29      end else begin
30          state[END_J] <= 1'b1;
31      end
32  end
```

For each FSM module instantiated, the corresponding Verilog code above should also be instantiated. To proceed with this step, it is necessary to take care of replacing the Js properly. In the above, the `J` should be replaced by the corresponding number $j$ of the $j^{th}$ cell. Note that `J+1` should be replaced by $((j+1) \mod n)$ and `J+2` by $((j+2) \mod n)$, where $n$ is the size of the ACM. For instance, the Verilog code obtained to control the access to cell number 0 is given by:

```
1   state[IDLE_0]:
2   begin
3       if (w_req) begin
4           w_ack <= 1'b0;
5           wreq <= 1'b1;
6           state[IDLE_0] <= 1'b0;
7           state[INIT_0] <= 1'b1;
8       end
9   end
10  state[INIT_0]:
```

```
11 begin
12     if (wack) begin
13         wreq <= 1'b0;
14         wdata <= w_data;
15         state[INIT_0] <= 1'b0;
16         state[END_0] <= 1'b1;
17     end
18 end
19 state[END_0]:
20 begin
21     if (!rd) begin
22         w_0 <= 1'b0;
23         w_1 <= 1'b1;
24         wsel <= 2;
25         w_ack <= 1'b1;
26         waddr <= 1;
27         state[END_0] <= 1'b0;
28         state[IDLE_1] <= 1'b1;
29     end else begin
30         state[END_0] <= 1'b1;
31     end
32 end
```

The source code obtained using the procedure described above implies two things:

1. All states of the FSM are enumerated and there is a state array that is one-hot encoded to indicate the current state;

2. There is a **case** statement in which the Verilog code above is inserted.

For instance, for the 3-cell RRBB ACM, the state enumeration of the writer engine sub-module is done by:

```
1 parameter   IDLE_0=0, INIT_0=1, END_0=2,
2             IDLE_1=3, INIT_1=4, END_1=5,
3             IDLE_2=6, INIT_2=7, END_2=8;
```

and the **case** statement is defined as below:

```
1 case (1'b1)
2     CASE BODY GENERATED FROM FSM
3 endcase
```

Observe that line 2 should be replaced by the proper case statements. The source code obtained for the reader engine is similar to the one obtained for the writer, and its generation follows the same idea. The main differences, as can be expected, are that the reader engine returns a data item to its

environment and that it does not blocks even in the absence of new non-read data. The Verilog template used to generate the reader engine is shown below.

```
state[IDLE_J]:
begin
    if (r_req) begin
        r_ack <= 1'b0;
        rreq <= 1'b1;
        state[IDLE_J] <= 1'b0;
        state[INIT_J] <= 1'b1;
    end
end
state[INIT_J]:
begin
    if (rack) begin
        rreq <= 1'b0;
        r_data <= rdata;
        state[INIT_J] <= 1'b0;
        state[END_J] <= 1'b1;
    end
end
state[END_J]:
begin
    if (!wr) begin
        r_J <= 1'b0;
        r_J+1 <= 1'b1;
        rsel <= J+2;
        r_ack <= 1'b1;
        raddr <= J+1;
        state[END_J] <= 1'b0;
        state[IDLE_J+1] <= 1'b1;
    end else begin
        r_ack <= 1'b1;
        raddr <= J;
        state[END_J] <= 1'b0;
        state[IDLE_J] <= 1'b1;
    end
end
```

As before, the Verilog code obtained will be embedded into the body of a **case** statement and it is necessary to generate a piece of code for each cell of the ACM. Again, all `J`s should be replaced by the corresponding number $j$ of the $j^{th}$ cell, `J+1` by $((j+1) \mod n)$ and `J+2` by $((j+2) \mod n)$, where $n$ is the size of the ACM.

## 7.5 An alternative approach

In [64], Silva describes an alternative method for the realization of a Petri net as a hardware artifact. That method can be applied for one-safe Petri nets, which are used in the present work, and it is based on the following principles:

1. Each place is implemented as a special type of flip-flop;
2. The flip-flop is activated when one of the transitions in the pre-set of the corresponding place is fired, and deactivated when one of the transitions in the post-set of the corresponding place is fired.

In other words, the places are implemented by an element with some memory, and the hardware implementation mimics the Petri net behavior.

An SR flip-flop is a digital circuit with memory to represent one bit. Typically it has two inputs named S (set) and R (reset) , besides the clock signal, and one output named Q as illustrated on Figure 7.10. The SR flip-flop may have priority on the set or on the reset signal.

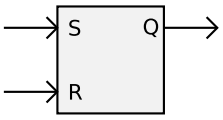



Figure 7.10: Set-Reset Flip-flop

The behavior of the SR flip-flop with priority on the set signal is given by the truth table in Table 7.1. If S is activated, then the output Q will also be activated. Q will remain activated even in the absence of S until the input R is activated. Note that R has effect only in the absence of S due to the fact that S has priority.

| S | R | Q |
|---|---|---|
| 0 | 0 | Q |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Table 7.1: Truth table for a SR flip-flop with priority on the set

This simple memory element can be used to implement a place with only one input and one output transition. For this reason the SR flip-flop is extended to implement places with more than one input and output transitions. This is done by using NAND gates as shown on Figure 7.11.

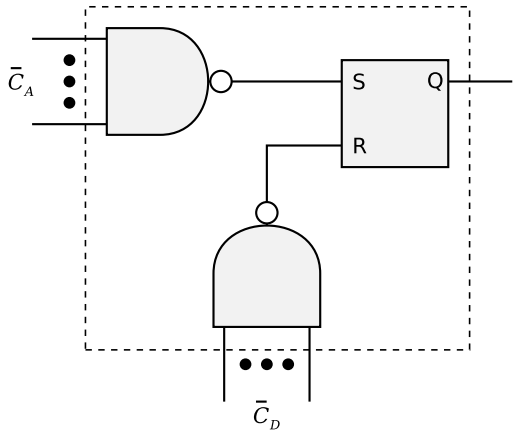



Figure 7.11: Memory cell build upon an SR flip-flop and NAND gates

$C_A$ gives the activation condition of the flip-flop, and $C_D$ gives the deactivation condition. In the Petri net notation $C_A$ and $C_D$ correspond to $\bullet p$ and $p\bullet$, where the $p$ is the place implemented by the flip-flop. $Q_p$ should be set to true if any of the activation conditions is satisfied. Observe that $S = \overline{\overline{C_{A_1}} \wedge \cdots \wedge \overline{C_{A_n}}}$, where $n = |\bullet p|$. This means that $S$ will be true if any input $C_A$ is true. This can be easily checked using the truth table of $S$ given by Table 7.2.

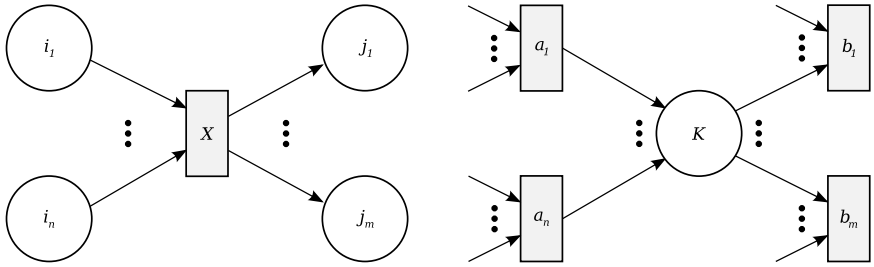
| $C_{A_1}$ | $\cdots$ | $C_{A_n}$ | S |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | x | 1 | 1 |
| 1 | x | 0 | 1 |
| 1 | 1 | 1 | 1 |

Table 7.2: Truth table for $S = \overline{\overline{C_{A_1}} \wedge \cdots \wedge \overline{C_{A_n}}}$

The same reasoning applies to the reset signal. Then it is necessary to specify how the flip-flops are activated and deactivated. In other words, it is

necessary to tell how to implement the behavior of the transitions and how to connect the implementations of transitions and places.

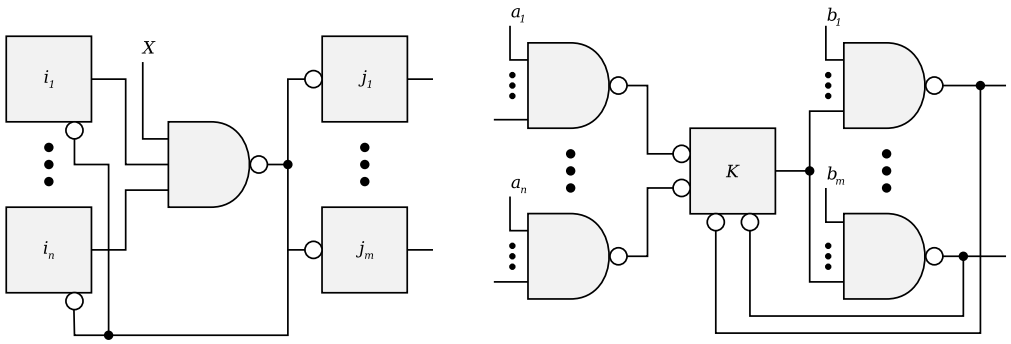
The activation of a flip-flop should happen when some of the input transitions of the place implemented by the flip-flop is fired. In the same way, the deactivation of the flip-flop occurs when an output transition of the place implemented by the flip-flop is fired. The transitions can be easily implemented by NAND or NOR logical gates.




(a) AND node (b) OR node

Figure 7.12: Typical PN structures

In Figure 7.12 typical structures of Petri net models are shown. These generic structures are well known in the literature as an AND node and an OR node. In short terms, these structures can be mapped into logic gates by replacing the places by the flip-flop cells as the one in Figure 7.11, and the transitions by NAND gates.




(a) AND node implementation (b) OR node implementation

Figure 7.13: Implementation of the PN models

As expected, each arc of the PN model corresponds to a wire connecting the logic elements. Besides that, the arcs linking a place $p$ to a transition $t$ also requires a wire from the NAND gate of $t$ to the flip-flop of $p$. This is

necessary due to the fact that a firing of $t$ is a cause of the deactivation of the flip-flop corresponding to $p$.

It is necessary to remember that the method discussed above can only be applied for one-safe Petri nets. However, it is possible to extend it in order to support $k$-limited Petri nets.

## 7.6 Conclusions

In this chapter the generation of hardware for the ACM re-reading protocol has been outlined. The basic structure of the implementation is defined by a set of block diagrams. These block diagrams have some gaps to be fulfilled to complete the implementation. These gaps correspond to the behavior of the writer and reader processes. It is shown how to obtain such behavior as an FSM specification and how these FSMs relate to the Petri net models of each process. Then, the generation of Verilog code from the FSMs is outlined.

The hardware generation method is not complete since it does address the overwriting policies. Clearly, this is a way this work can be extended. However, from an engineering point of view, the method is a proper proof of concept which indicates that the same approach can be used to obtain the implementation of more complex protocols.

# Chapter 8

# Conclusions and future work

In this thesis Asynchronous Communication Mechanisms (ACMs) have been studied, and two methods for the automatic generation of ACMs have been proposed. The first method is based on the construction of a state graph specification that is synthesized into a Petri. The second approach skips the state graph generation and attempts to construct the Petri net model of the ACM directly. In both cases, the Petri net models are used to derive software and hardware ACM implementations.

Since most ACMs tend to use single bit control variables, besides being time consuming, the task of building ACMs manually is prone to errors when the size of the ACM grows. This is mainly due to the number of control variables needed to implement an ACM. The importance of automatic methods for the synthesis of ACMs relies on the guarantee that the obtained artifact is free from errors and on how fast the implementation can be obtained. In this work it has been shown how to obtain ACM implementations with a reasonable guarantee that the artifacts preserve the coherence and freshness properties.

The state graph based approach is described in Chapter 3. There, the automation of two key steps in ACM synthesis is detailed. These are: i) deriving of an interleaving state graph specification given a functional specification; and ii) the generation of an ACM Petri net model in the form of an independent state machine using unidirectional shared variables. Previously, these steps were the most tedious and time-consuming tasks in a manual synthesis process, and, although suggested to be "automatable" because of their systematic and step-by-step nature, they were never shown to be so conclusively. We have applied this method to a number of "standard" ACM types for writing and reading multi-cell buffers.

In Chapter 4 another approach to the automatic synthesis of ACMs is introduced. This method is based on the use of modules for the construction

of Petri net models that can be verified against a more abstract specification. Firstly, the behavior of re-reading and overwriting ACMs was formally defined and the properties they should satisfy were described by CTL formulae. 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.

Compared to the state graph specification approach, the method of generating Petri net models has the disadvantage of requiring the use of model checking techniques to guarantee the correctness of the models generated. In the first approach based on the state graph specification and ACM regions, it was guaranteed by construction. However, the cost of executing the ACM regions algorithm is too high, and when it becomes limited by the state space explosion problem, no Petri net model could be generated and synthesis fails. In the second approach, state space explosion is limited to the verification of the Petri net model. This step is off the main design flow, as outlined in Figure 4.1. Thus we could generate source code from the Petri net 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 verification of the Petri net models is detailed in Chapter 5. Firstly, the mode abstract ACM behaviors described in Definitions 4.1 and 4.2 are modeled as transition systems and verified to satisfy coherence and freshness properties. Secondly, the Petri net model is transformed into a transition system and is verified to be a refinement of the more abstract model. It was possible to verify the abstract models of ACM of considerable size. However, the refinement verification of overwriting ACMs is still very limited. It has been observed that even for ACMs of small size, the state space explosion problem causes the verification to fail. This happens mainly due to the amount of variables needed to correctly implement the overwriting ACMs, which is considerable bigger than for re-reading ones.

Finally, the synthesis of ACMs implementations is detailed. Chapter 6 details the generation of C++ source code for re-reading and overwriting policies. The only assumptions made about the operating system in which the ACM will execute are: i) the existence of some inter-process communication mechanism to allow a process to send a signal to another process; and ii) the support of shared memory segments that can be accessed by two processes. Then the generation of C++ source code for POSIX compliant operating systems is detailed.

In Chapter 7 the synthesis of hardware is detailed for the re-reading policy. Firstly, the basic hardware structure is described as a set of block diagrams.

These block diagrams are used as templates in which the behavior of each ACM process will replace some specific blocks. Then the generation of Verilog code, which can be realized in hardware, for each ACM process is detailed. The Verilog code replaces the corresponding blocks in the diagram, which concludes the synthesis procedure.

Since model checking results for the overwriting results were not very exciting, formal verification of ACM models is one direction in which this work can be extended. One possibility is to work on the Petri net model in order to reduce the amount of control variables needed. This should make it possible to verify bigger models, but this only alleviates the state explosion problem. Another possibility is to apply induction proof methods.

Also, the generation of hardware for the overwriting policies has not been addressed. This is another way of extending this work. Furthermore, it is necessary to extend the hardware and software generation mechanism to more realistic scenarios, in which the details of the available technologies are considered. One possibility for the hardware generation schema is the development of asynchronous memory modules which make use of the protocols and methods discussed in this work. The application of the software generation schema to be used in the Linux kernel in order to implement asynchronous system call is a possible practical application of ACMs.

Finally, all protocols addressed here only allow one writer and one reader. It is necessary further investigation to obtain protocols that allows multiple writers and/or readers. Besides that, the use of the shared memory in both directions (i.e. both processes can read it and write into it) may also be object of future investigation.

# Appendix A

# SVM samples

On this appendix the SMV model of a 3-cell RRBB ACM is included. A complete set of examples can be found at `http://www.dee.ufcg.edu.br/~kyller/acms/smv-samples.tar.bz2`.

## A.1 3-cell RRBB ACM SMV sample

```
1  -- Generated by ACMgen
2  -- Linux 2.6.26-1-686 at Fri Sep 12 09:57:27 BRT 2008
3  #define true        1
4  #define false       0
5
6  #define ACM_SIZE    3
7  typedef ACM         0..(ACM_SIZE - 1);
8
9  module writer_p(writer, reader, rd_data, wr_cont, acm, wr_data,
       acm2) {
10
11     case {
12         writer = idle & wr_cont < ACM_SIZE:
13         {
14             next(writer) := accessing;
15             next(reader) := reader;
16             next(rd_data) := rd_data;
17             next(wr_cont) := wr_cont;
18             for (i = 0; i < ACM_SIZE; i = i + 1) {
19                 if (i = wr_cont) {
20                     next(acm[i]) := wr_data;
21                 } else {
22                     next(acm[i]) := acm[i];
23                 }
24             }
```

```
25              next(acm2) := acm;
26          }
27
28          writer = accessing:
29          {
30              next(writer) := idle;
31              next(reader) := reader;
32              next(rd_data) := rd_data;
33              next(wr_cont) := wr_cont + 1;
34              next(acm) := acm;
35              next(acm2) := acm;
36          }
37      }
38  }
39
40  module reader_p(writer, reader, rd_data, wr_cont, acm, wr_data,
        acm2) {
41
42      case {
43
44          reader = idle:
45          {
46              next(writer) := writer;
47              next(reader) := accessing;
48              next(rd_data) := acm[0];
49              next(wr_cont) := wr_cont;
50              next(acm) := acm;
51              next(acm2) := acm;
52          }
53
54          reader = accessing & wr_cont = 1:
55          {
56              next(writer) := writer;
57              next(reader) := idle;
58              next(rd_data) := rd_data;
59              next(wr_cont) := wr_cont;
60              next(acm) := acm;
61              next(acm2) := acm;
62          }
63
64          reader = accessing & wr_cont > 1:
65          {
66              next(writer) := writer;
67              next(reader) := idle;
68              next(rd_data) := rd_data;
69              next(wr_cont) := wr_cont - 1;
70              for (i = 0; i < ACM_SIZE-1; i = i + 1) {
71                  next(acm[i]) := acm[i + 1];
72              }
```

```
73              next(acm2) := acm;
74          }
75      }
76  }
77
78  module main() {
79
80      -- variables declarations
81      writer, reader: {idle,accessing};
82
83      acm, acm2: array ACM of boolean;
84
85      wr_data, rd_data: boolean;
86      wr_cont: 0..ACM_SIZE;
87
88      -- Specification
89      layer rrbb: {
90
91          init(writer) := idle;
92          init(reader) := idle;
93
94          init(acm[0]) := wr_data;
95          init(wr_cont) := 1;
96          init(acm2) := acm;
97          next(acm2) := acm;
98
99          wrp: process writer_p(writer, reader, rd_data, wr_cont,
                acm, wr_data, acm2);
100         rdp: process reader_p(writer, reader, rd_data, wr_cont,
                acm, wr_data, acm2);
101
102         case {
103             wr_cont < ACM_SIZE | writer = accessing: {
104                 wrp.running := {0,1};
105                 rdp.running := ~wrp.running;
106             }
107
108             default: {
109                 wrp.running := 0;
110                 rdp.running := 1;
111             }
112         }
113     }
114
115     -- FAIRNESS CONSTRAINTS
116     FAIRNESS wrp.running;
117     FAIRNESS rdp.running;
118
119     -- CTL PROPERTIES
```

```
120     ----> COHERENCE
121     SPEC AG(reader = accessing -> (rd_data = acm[0] & wr_cont >
            0));
122
123     ----> FRESHNESS
124     -- sequencing properties for wr_cont = 1
125     SPEC AG(wr_cont = 1 ->
126             AX((wr_cont >= 1 & acm[0] = acm2[0])));
127     -- sequencing properties for wr_cont = 2
128     SPEC AG(wr_cont = 2 ->
129             AX((wr_cont >= 2 & acm[0..1] = acm2[0..1]) ||
130                 (wr_cont = 1  & acm[0] = acm2[1])));
131     -- sequencing properties for wr_cont = 3
132     SPEC AG(wr_cont = 3 ->
133             AX((wr_cont >= 3 & acm[0..2] = acm2[0..2]) ||
134                 (wr_cont = 2  & acm[0..1] = acm2[1..2])));
135 }
```

# Appendix B

# C++ samples

On this appendix the C++ source code the writer process of a 3-cell RRBB ACM is included. A complete set of examples can be found at `http://www.dee.ufcg.edu.br/~kyller/acms/cpp-samples.tar.bz2`.

## B.1 3-cell RRBB ACM C++ sample

### B.1.1 Writer process .h file

```
1  using namespace std;
2
3  #include <errno.h>
4  #include <signal.h>
5  #include <stdio.h>
6  #include <sys/ipc.h>
7  #include <sys/shm.h>
8  #include <sys/types.h>
9  #include <unistd.h>
10
11 #include <iostream>
12
13 #ifndef Writer_h
14 #define Writer_h
15
16 #define ACM_SIZE        3
17
18 #define SHMKEY          ((key_t) 21000)
19 #define SHMKEY1         ((key_t) 21100)
20 #define PERMS           0666
21
22 #define acm_t           char
23
```

```
24 class Writer {
25
26     public:
27
28         acm_t *shm_data;
29         long long int shm_id;
30         pid_t pair_pid;
31
32         // writer variables
33         bool wi0;
34         bool wi1;
35         bool pwi0;
36         bool *we0;
37         int we0_shmid;
38         bool *wne0;
39         int wne0_shmid;
40         bool *we1;
41         int we1_shmid;
42         bool *wne1;
43         int wne1_shmid;
44         bool *rne1;
45         int rne1_shmid;
46         bool wi2;
47         bool pwi1;
48         bool *we2;
49         int we2_shmid;
50         bool *wne2;
51         int wne2_shmid;
52         bool *rne2;
53         int rne2_shmid;
54         bool pwi2;
55         bool *rne0;
56         int rne0_shmid;
57
58         Writer(pid_t);
59         ~Writer();
60
61         void Write(acm_t);
62         void Write1(acm_t);
63         void Send(acm_t);
64         void Lambda(void);
65
66     private:
67
68         bool InitLocalShv();
69         bool InitExternalShv();
70         void DtLocalShv();
71         void DtExternalShv();
72         static void SignalHandler(int);
```

```
73 };
74 #endif // Writer_h
```

### B.1.2 Writer process .cpp file

```
1  using namespace std;
2
3  #include "Writer.h"
4
5  extern int errno;
6
7  Writer::Writer(pid_t pid) {
8
9      pair_pid = pid;
10     signal(SIGCONT, SignalHandler);
11
12     if ((shm_id = shmget(SHMKEY, sizeof(acm_t) * ACM_SIZE, PERMS
           | IPC_CREAT)) < 0) {
13
14         cerr << "Error creating Writer. Cannot exec shmget()" <<
               endl;
15         exit(errno);
16     }
17
18     if ((shm_data = (acm_t *) shmat(shm_id, (acm_t *) 0, 0)) == (
           acm_t *) -1) {
19
20         cerr << "Error creating Writer. Cannot exec shmat()" <<
               endl;
21         exit(errno);
22     }
23
24     *shm_data = '-';
25
26     // writer variables
27     wi0 = false;
28     wi1 = true;
29     pwi0 = false;
30     wi2 = false;
31     pwi1 = false;
32     pwi2 = false;
33
34     InitLocalShv();
35     InitExternalShv();
36 }
37
38 Writer::~Writer() {
39
40     if (shmdt(shm_data)) {
```

```
41
42          cout << "Error destroying Writer. Cannont exec shmdt()"
                << endl;
43          exit(errno);
44      }
45
46      DtExternalShv();
47      DtLocalShv();
48  }
49
50  bool Writer::InitLocalShv(void) {
51
52      if ((we0_shmid = shmget(SHMKEY1 + 16, sizeof(bool), PERMS |
            IPC_CREAT)) < 0) {
53
54          cout << "Error creating we0_shmid. Cannot exec shmget()"
                << endl;
55          exit(errno);
56      }
57      if ((we0 = (bool *) shmat(we0_shmid, (bool *) 0, 0)) == (bool
            *) -1) {
58
59          cout << "Error creating we0. Cannot exec shmat()" << endl
                ;
60          exit(errno);
61      }
62      *we0 = 0;
63
64      if ((wne0_shmid = shmget(SHMKEY1 + 17, sizeof(bool), PERMS |
            IPC_CREAT)) < 0) {
65
66          cout << "Error creating wne0_shmid. Cannot exec shmget()"
                 << endl;
67          exit(errno);
68      }
69      if ((wne0 = (bool *) shmat(wne0_shmid, (bool *) 0, 0)) == (
            bool *) -1) {
70
71          cout << "Error creating wne0. Cannot exec shmat()" <<
                endl;
72          exit(errno);
73      }
74      *wne0 = 1;
75
76      if ((we1_shmid = shmget(SHMKEY1 + 7, sizeof(bool), PERMS |
            IPC_CREAT)) < 0) {
77
78          cout << "Error creating we1_shmid. Cannot exec shmget()"
                << endl;
```

```
79          exit(errno);
80      }
81      if ((we1 = (bool *) shmat(we1_shmid, (bool *) 0, 0)) == (bool
            *) -1) {
82
83          cout << "Error creating we1. Cannot exec shmat()" << endl
                ;
84          exit(errno);
85      }
86      *we1 = 1;
87
88      if ((wne1_shmid = shmget(SHMKEY1 + 8, sizeof(bool), PERMS |
            IPC_CREAT)) < 0) {
89
90          cout << "Error creating wne1_shmid. Cannot exec shmget()"
                << endl;
91          exit(errno);
92      }
93      if ((wne1 = (bool *) shmat(wne1_shmid, (bool *) 0, 0)) == (
            bool *) -1) {
94
95          cout << "Error creating wne1. Cannot exec shmat()" <<
                endl;
96          exit(errno);
97      }
98      *wne1 = 0;
99
100     if ((we2_shmid = shmget(SHMKEY1 + 13, sizeof(bool), PERMS |
            IPC_CREAT)) < 0) {
101
102         cout << "Error creating we2_shmid. Cannot exec shmget()"
                << endl;
103         exit(errno);
104     }
105     if ((we2 = (bool *) shmat(we2_shmid, (bool *) 0, 0)) == (bool
            *) -1) {
106
107         cout << "Error creating we2. Cannot exec shmat()" << endl
                ;
108         exit(errno);
109     }
110     *we2 = 0;
111
112     if ((wne2_shmid = shmget(SHMKEY1 + 14, sizeof(bool), PERMS |
            IPC_CREAT)) < 0) {
113
114         cout << "Error creating wne2_shmid. Cannot exec shmget()"
                << endl;
115         exit(errno);
```

```
116     }
117     if ((wne2 = (bool *) shmat(wne2_shmid, (bool *) 0, 0)) == (
            bool *) -1) {
118 
119         cout << "Error creating wne2. Cannot exec shmat()" <<
                endl;
120         exit(errno);
121     }
122     *wne2 = 1;
123 
124 }
125 
126 bool Writer::InitExternalShv(void) {
127 
128     while ((rne1_shmid = shmget((SHMKEY1 + 6), sizeof(bool), 0))
            < 0) {
129 
130         cout << "Error creating rne1_shmid. Cannot exec shmget()"
                << endl;
131         kill(pair_pid, SIGCONT);          sleep(1);//      exit(
                errno);
132     }
133     while ((rne1 = (bool *) shmat(rne1_shmid, (bool *) 0, 0)) ==
            (bool *) -1) {
134 
135         cout << "Error creating rne1. Cannot exec shmat()" <<
                endl;
136         kill(pair_pid, SIGCONT);          sleep(1);//      exit(
                errno);
137     }
138     while ((rne2_shmid = shmget((SHMKEY1 + 12), sizeof(bool), 0))
            < 0) {
139 
140         cout << "Error creating rne2_shmid. Cannot exec shmget()"
                << endl;
141         kill(pair_pid, SIGCONT);          sleep(1);//      exit(
                errno);
142     }
143     while ((rne2 = (bool *) shmat(rne2_shmid, (bool *) 0, 0)) ==
            (bool *) -1) {
144 
145         cout << "Error creating rne2. Cannot exec shmat()" <<
                endl;
146         kill(pair_pid, SIGCONT);          sleep(1);//      exit(
                errno);
147     }
148     while ((rne0_shmid = shmget((SHMKEY1 + 4), sizeof(bool), 0))
            < 0) {
149 
```

```
150             cout << "Error creating rne0_shmid. Cannot exec shmget()"
                    << endl;
151             kill(pair_pid, SIGCONT);          sleep(1);//      exit(
                    errno);
152         }
153         while ((rne0 = (bool *) shmat(rne0_shmid, (bool *) 0, 0)) ==
                (bool *) -1) {
154
155             cout << "Error creating rne0. Cannot exec shmat()" <<
                    endl;
156             kill(pair_pid, SIGCONT);          sleep(1);//      exit(
                    errno);
157         }
158     }
159
160     void Writer::DtLocalShv(void) {
161
162         if (shmdt(we0)) {
163
164             cout << "Error destroying we0. Cannot exec shmdt()" <<
                    endl;
165             exit(errno);
166         }
167
168         if (shmdt(wne0)) {
169
170             cout << "Error destroying wne0. Cannot exec shmdt()" <<
                    endl;
171             exit(errno);
172         }
173
174         if (shmdt(we1)) {
175
176             cout << "Error destroying we1. Cannot exec shmdt()" <<
                    endl;
177             exit(errno);
178         }
179
180         if (shmdt(wne1)) {
181
182             cout << "Error destroying wne1. Cannot exec shmdt()" <<
                    endl;
183             exit(errno);
184         }
185
186         if (shmdt(we2)) {
187
188             cout << "Error destroying we2. Cannot exec shmdt()" <<
                    endl;
```

```
189             exit(errno);
190         }
191 
192         if (shmdt(wne2)) {
193 
194             cout << "Error destroying wne2. Cannot exec shmdt()" <<
                    endl;
195             exit(errno);
196         }
197 
198     }
199 
200     void Writer::DtExternalShv(void) {
201 
202         if (shmdt(rne1)) {
203 
204             cout << "Error destroying rne1. Cannot exec shmdt()" <<
                    endl;
205             exit(errno);
206         }
207 
208         if (shmctl(rne1_shmid, IPC_RMID, (struct shmid_ds *) 0) < 0)
                {
209 
210             cout << "Error destroying rne1_shmid. Cannot exec shmctl
                    ()" << endl;
211             exit(errno);
212         }
213 
214         if (shmdt(rne2)) {
215 
216             cout << "Error destroying rne2. Cannot exec shmdt()" <<
                    endl;
217             exit(errno);
218         }
219 
220         if (shmctl(rne2_shmid, IPC_RMID, (struct shmid_ds *) 0) < 0)
                {
221 
222             cout << "Error destroying rne2_shmid. Cannot exec shmctl
                    ()" << endl;
223             exit(errno);
224         }
225 
226         if (shmdt(rne0)) {
227 
228             cout << "Error destroying rne0. Cannot exec shmdt()" <<
                    endl;
229             exit(errno);
```

```
230     }
231
232     if (shmctl(rne0_shmid, IPC_RMID, (struct shmid_ds *) 0) < 0)
            {
233
234         cout << "Error destroying rne0_shmid. Cannot exec shmctl
                ()" << endl;
235         exit(errno);
236     }
237
238 }
239
240 void Writer::Write(acm_t val) {
241
242     Send(val);
243     Lambda();
244 }
245
246 void Writer::Write1(acm_t val) {
247
248     Lambda();
249     Send(val);
250 }
251
252 void Writer::Send(acm_t val) {
253
254     if (wi0 == true) {
255         // wr0
256         wi0 = false;
257         pwi0 = true;
258         *(shm_data + 0) = val;
259     } else if (wi1 == true) {
260         // wr1
261         wi1 = false;
262         pwi1 = true;
263         *(shm_data + 1) = val;
264     } else if (wi2 == true) {
265         // wr2
266         wi2 = false;
267         pwi2 = true;
268         *(shm_data + 2) = val;
269     }
270 }
271
272 void Writer::Lambda(void) {
273
274     while (true) {
275
276         // l0_1
```

```
277         if (*we0 == true && *wne1 == true && pwi0 == true && *
                rne1 == true) {
278
279             *we0 = false;
280             *wne1 = false;
281             pwi0 = false;
282             wi1 = true;
283             *wne0 = true;
284             *we1 = true;
285             break;
286         }
287         // l1_2
288         if (*we1 == true && *wne2 == true && pwi1 == true && *
                rne2 == true) {
289
290             *we1 = false;
291             *wne2 = false;
292             pwi1 = false;
293             wi2 = true;
294             *wne1 = true;
295             *we2 = true;
296             break;
297         }
298         // l2_0
299         if (*we2 == true && *wne0 == true && pwi2 == true && *
                rne0 == true) {
300
301             *we2 = false;
302             *wne0 = false;
303             pwi2 = false;
304             wi0 = true;
305             *wne2 = true;
306             *we0 = true;
307             break;
308         }
309
310         pause();
311     }
312 }
313
314 void Writer::SignalHandler(int signo) {
315
316     if (signo == SIGCONT) {}
317
318     return;
319 }
```

# Appendix C

# Verilog samples

On this appendix the Verilog code for the writer process of a 3-cell RRBB ACM is included. A complete set of examples can be found at `http://www.dee.ufcg.edu.br/~kyller/acms/verilog-samples.tar.bz2`.

## C.1 3-cell RRBB ACM Verilog sample

```
1  module writer_engine(
2      clock,
3      ack,
4      ereq,
5      rd,
6      edata,
7      cell0,
8      cell1,
9      cell2,
10     req,
11     eack,
12     addr,
13     data,
14     sel,
15     reset
16 );
17
18 parameter
19     IDLE0 = 0,
20     INIT0 = 1,
21     END0 = 2,
22     IDLE1 = 3,
23     INIT1 = 4,
24     END1 = 5,
25     IDLE2 = 6,
```

```
26      INIT2 = 7,
27      END2 = 8;
28
29  input    clock;
30  input    ack;
31  input    ereq;
32  input    rd;
33  input    [31:0] edata;
34  input    reset;
35  output   cell0;
36  output   cell1;
37  output   cell2;
38  output   req;
39  output   eack;
40  output   [1:0] addr;
41  output   [31:0] data;
42  output   [1:0] sel;
43
44  reg cell0;
45  reg cell1;
46  reg cell2;
47  reg req;
48  reg eack;
49  reg [1:0] addr;
50  reg [31:0] data;
51  reg [1:0] sel;
52
53  reg [8:0] state;
54
55  always @(posedge clock or posedge reset) begin
56      if (reset) begin
57          cell0 <= 1'b0;
58          cell1 <= 1'b1;
59          cell2 <= 1'b0;
60          sel <= 2;
61          req <= 1'b0;
62          eack <= 1'b0;
63          addr <= 1;
64          data <= edata;
65          state <= 9'b0;
66          state[IDLE1] <= 1'b1;
67      end else begin
68          case (1'b1)
69              state[IDLE0]: begin
70                  if (ereq) begin
71                      eack <= 1'b0;
72                      req <= 1'b1;
73                      state[IDLE0] <= 1'b0;
74                      state[INIT0] <= 1'b1;
```

```
75                  end
76              end
77              state[INIT0]: begin
78                  if (ack) begin
79                      req <= 1'b0;
80                      data <= edata;
81                      state[INIT0] <= 1'b0;
82                      state[END0] <= 1'b1;
83                  end
84              end
85              state[END0]: begin
86                  if (!rd) begin
87                      cell0 <= 1'b0;
88                      cell1 <= 1'b1;
89                      sel <= 2;
90                      eack <= 1'b1;
91                      addr <= 1;
92                      state[END0] <= 1'b0;
93                      state[IDLE1] <= 1'b1;
94                  end else begin
95                      state[END0] <= 1'b1;
96                  end
97              end
98              state[IDLE1]: begin
99                  if (ereq) begin
100                     eack <= 1'b0;
101                     req <= 1'b1;
102                     state[IDLE1] <= 1'b0;
103                     state[INIT1] <= 1'b1;
104                 end
105             end
106             state[INIT1]: begin
107                 if (ack) begin
108                     req <= 1'b0;
109                     data <= edata;
110                     state[INIT1] <= 1'b0;
111                     state[END1] <= 1'b1;
112                 end
113             end
114             state[END1]: begin
115                 if (!rd) begin
116                     cell1 <= 1'b0;
117                     cell2 <= 1'b1;
118                     sel <= 0;
119                     eack <= 1'b1;
120                     addr <= 2;
121                     state[END1] <= 1'b0;
122                     state[IDLE2] <= 1'b1;
123                 end else begin
```

```
124                     state[END1] <= 1'b1;
125                 end
126             end
127             state[IDLE2]: begin
128                 if (ereq) begin
129                     eack <= 1'b0;
130                     req <= 1'b1;
131                     state[IDLE2] <= 1'b0;
132                     state[INIT2] <= 1'b1;
133                 end
134             end
135             state[INIT2]: begin
136                 if (ack) begin
137                     req <= 1'b0;
138                     data <= edata;
139                     state[INIT2] <= 1'b0;
140                     state[END2] <= 1'b1;
141                 end
142             end
143             state[END2]: begin
144                 if (!rd) begin
145                     cell2 <= 1'b0;
146                     cell0 <= 1'b1;
147                     sel <= 1;
148                     eack <= 1'b1;
149                     addr <= 0;
150                     state[END2] <= 1'b0;
151                     state[IDLE0] <= 1'b1;
152                 end else begin
153                     state[END2] <= 1'b1;
154                 end
155             end
156         endcase
157     end
158 end
159
160 endmodule
```

# Bibliography


[1] ITRS 2007 edition. `http://www.itrs.net/Links/2007ITRS/Home2007.htm`, 2007.

[2] Software/dbus. `http://www.freedesktop.org/wiki/Software/dbus`, 2009.

[3] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[4] Maurice J. Bach. *The design of the UNIX operating system.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[5] M. Ben-Ari. *Principles of Concurrent Programming.* Prentice Hall International, 1982.

[6] Allan Cheng, Søren Christensen, and Kjeld Høyer Mortensen. Model checking coloured petri nets exploiting strongly connected components. Technical report, Computer Science Department, Aarhus University, Aarhus (Denmark), March 1997.

[7] Søren Christensen and Kjeld Høyer Mortensen. *Design/CPN ASK-CTL Manual*, 1996.

[8] Edmund M. Clarke. *The Birth of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 1–26. Springer Berlin / Heidelberg, July 2008.

[9] Edmund M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights*, volume 131 of *Lectures Notes in Computer Science*, New York, USA, May 1981. Springer-Verlag.

[10] Edmund M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[11] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5), September 1994.

[12] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking. *Springer-Verlag Nato ASI Series F*, 152, 1996. a survey on model checking, abstraction and composition.

[13] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, USA, 1999.

[14] Edmund M. Clarke, David E. Long, and Kenneth L. McMillan. Compositional model checking. In *Proceedings of the Fourth Anual Symposium on Logic in Computer Science*, pages 353–362, Pacific Grove, California (USA), 1989.

[15] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[16] 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.

[17] Jordi Cortadella, Michael Kishinevsky, Luciano Lavagno, and Alex Yakovlev. Deriving Petri nets from finite transition systems. *IEEE Transactions on Computers*, 47(8):859–882, August 1998.

[18] Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[19] Bruce Eckel. *Thinking in C++, Volume I: Introduction to Standard C++, Second Edition*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[20] Bruce Eckel, Chuck D. Allison, and Chuck Allison. *Thinking in C++, Vol. 2.* Pearson Education, 2003.

[21] E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 995–1072. Elsevier Science Publisher B.V., 1990.

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

[23] Claude Girault and Rüdiger Valk, editors. *Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications.* Springer-Verlag, Berlin, Germany, 2003.

[24] Kyller Gorgônio and Jordi Cortadella. Hardware synthesis for asynchronous communications mechanisms. In *International Conference of the Chilean Computer Science Society (SCCC 2008)*, pages 135–143, Punta Arenas, Chile, November 2008. IEEE Computer Society.

[25] Kyller Gorgônio, Jordi Cortadella, and Fei Xia. A compositional method for the synthesis of asynchronous communication mechanisms. In Jetty Kleijn and Alex Yakovlev, editors, *28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency (ICATPN 2007)*, number 4546 in Lecture Notes in Computer Science, pages 144–163. Springer-Verlag Berlin Heidelberg, 2007.

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

[27] Kyller Gorgônio and Fei Xia. Modeling and verifying asynchronous communication mechanisms using coloured petri nets. In *Proceedings of the Eighth International Conference on Application of Concurrency to System Design (ACSD'08)*, pages 138–147, Xi'an, China, June 2008. IEEE Computer Society.

[28] Kyller Gorgônio and Fei Xia. Modeling and verifying asynchronous communication mechanisms using coloured petri nets. Technical Report Series NCL-EECE-MSD-TR-2008-127, Newcastle University, Newcastle upon Tyne, NE1 7RU, UK, March 2008.

[29] 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.

[30] Per Brinch Hansen. The programming language concurrent pascal. *IEEE Software Engineering*, SE-1(2):199–207, 1975.

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

[32] D. Hauschildt. A petri net implementation. technical report, Fachbereich Informatik, Universität Hamburg, Hamburg, Germany, 1987.

[33] Neil Henderson and Stephen E. Paynter. The formal classification and verification of simpson's 4-slot asynchronous communication mechanism. Technical Report 756, University of Newcastle upon Tyne, 2002.

[34] C. A. R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[35] C. A. R. Hoare. *Communicating Sequencial Processes.* Prentice Hall International Series in Computer Science. Prentice Hall International, 1985.

[36] G. E. Hughes and M. J. Creswell. *Introduction to Modal Logic.* Methuen, London (UK), 1977.

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

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

[39] 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.

[40] G. Jones and M. Goldsmith. *Programming in Occam 2.* Prentice Hall International, 1988.

[41] Clinton Kelly IV, Virantha Ekanayake, and Rajit Manohar. Snap: A sensor-network asynchronous processor. In *Proceedings of the 9th International Symposium on Asynchronous Circuits and Systems*, page 24. IEEE Computer Society, 2003.

[42] Lindsay Kleeman and Antonio Cantoni. On the unavoidability of metastable behavior in digital systems. *IEEE Transactions on Computers*, 36(1):109–112, 1987.

[43] F. Kordon, P. Estraillier, and R. Card. Rapid ada prototyping: principles and example of a complex application. In *Proceedings of the Ninth International Conference on Computers and Communications*, pages 453–460, Piscataway, NJ, USA, March 1990. IEEE Service Center.

[44] Leslie Lamport. The mutual exclusion problem: Part i — a theory of interprocess communication. *Journal of the ACM (JACM)*, 33(2):313–326, 1986.

[45] Leslie Lamport. On interprocess communication — part I: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

[46] Leslie Lamport. On interprocess communication — part II: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[47] B. Lin. Software synthesis of process-based concurrent programs. In *35th ACM/IEEE Design Automation Conference*, June 1998.

[48] Leonard R. Marino. General theory of metastable operation. *IEEE Transactions on Computers*, 30(2):107–115, 1981.

[49] Kenneth L. McMillan. The SMV system. Technical Report CMU-CS-92-131, 1992.

[50] Kenneth L. McMillan. *Getting started with SMV*. Cadence Berkley Labs, Cadence Design Systems, Berkeley, CA, USA, March 1999.

[51] 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.

[52] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980.

[53] Kjeld H. Mortensen. Automatic code generation method based on coloured petri net models applied on an access control system. In M. Nielsen and D. Simpson, editors, *Lecture Notes in Computer Science: 21st International Conference on Application and Theory of Petri Nets (ICATPN 2000), Aarhus, Denmark, June 2000*, volume 1825, pages 367–386. Springer-Verlag, 2000.

[54] David E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, April 1959.

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

[56] Mogens Nielsen, Grzegorz Rozenberg, and P. S. Thiagarajan. Elementary transition systems. *Theoretical Computer Science*, 96(1):3–33, 1992.

[57] James L. Peterson. *Petri Net Theory and Modeling of Systems*. Prentice Hall, 1981.

[58] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, October 1977. IEEE, IEEE Computer Society.

[59] 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.

[60] Eric S. Raymond. *The Art of UNIX Programming*. Addison-Wesley Professional, September 2003.

[61] M. Raynal and D. Beeson. *Algorithms for mutual exclusion*. MIT Press, 1986.

[62] Charles L. Seitz. System timing. In Carver Mead and Lynn Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

[63] Marco Sgroi, Luciano Lavagno, Yosinori Watanabe, and Alberto Sangiovanni-Vincentelli. Synthesis of embedded software using free-choice petri nets. In *36th ACM/IEEE Design Automation Conference*, June 1999.

[64] Manuel Silva. *Las Redes de Petri en la Automática y la Informática*. Editorial AC, Madrid, Spain, 1985.

[65] Hugo R. Simpson. Four-slot fully asynchronous communication mechanism. *IEE Proceedings on Computers and Digital Techniques*, 137 Part E(1):17–30, January 1990.

[66] Hugo R. Simpson. Correctness analysis for class of asynchronous communication mechanisms. *IEE Proceedings on Computers and Digital Techniques*, 139 Part E(1):35–49, January 1992.

[67] Hugo R. Simpson. Methodological and notational conventions in doris real-time networks. Dynamics Division, BAe, February 1994.

[68] Hugo R. Simpson. Role model analysis of an asynchronous communication mechanism. *IEE Proceedings on Computers and Digital Techniques*, 144(4):232–240, July 1997.

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

[70] Hugo R. Simpson and Eric Campbell. Real-time network architecture: principles and practice. In *Proceedings AINT 2000. Asynchronous INTerfaces: tools, techniques and implementations*, page p.5 and handouts, The Netherlands, July 2000. TU Delft.

[71] William Stallings. *Operating systems.* Macmillan Publishing Co., Inc., Indianapolis, IN, USA, 1992.

[72] W. Richard Stevens. *Advanced programming in the UNIX environment.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[73] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. *UNIX Network Programming, Vol. 1.* Pearson Education, 2003.

[74] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Professional, special 3rd edition edition, February 2000.

[75] Andrew S. Tanenbaum. *Modern operating systems.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[76] Dirk Taubner. On the implementation of petri nets. pages 418–439, 1988.

[77] Donald E. Thomas and Philip R. Moorby. *The Verilog hardware description language (4th ed.).* Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[78] Fei Xia and Ian Clark. Algorithms for signal and message asynchronous communication mechanisms and their analysis. In *Proceedings of the*

*2nd International Conference on Applications of Concurrency to System Design*, pages 65–74, Newcastle upon Tyne (UK), June 2001. IEEE Computer Society.

[79] Fei Xia and Ian Clark. Algorithms for signal and message asynchronous communication mechanisms and their analysis. *Fundamenta Informaticae*, 50(2):205–222, April 2002.

[80] Fei Xia, Fei Hao, Ian Clark, Alex Yakovlev, and E. Graeme Chester. Buffered asynchronous communication mechanisms. *Fundamenta Informaticae*, 70(1,2):155–170, March 2006.

[81] 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.

[82] Fei Xia, Alex Yakovlev, Ian G. Clark, and Delong Shang. Data communication in systems with heterogeneous timing. *IEEE Micro*, 22(6):58–69, 2002.

[83] Fei Xia, Alex Yakovlev, Delong Shang, Alexandre Bystrov, Albert Koelmans, and David Kinniment. Asynchronous communication mechanisms using self-timed circuits. In *ASYNC'00: Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, page 150, Washington, DC, USA, 2000. IEEE Computer Society.

[84] 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.

[85] 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.

[86] Alex Yakovlev, Fei Xia, and Delong Shang. Synthesis and implementation of a signal-type asynchronous data communication mechanism. In *ASYNC'01: Proceedings of the 7th International Symposium on Asynchronous Circuits and Systems*, pages 127–136. IEEE Computer Society, 2001.