Automating Synthesis of Asynchronous Communication Mechanisms

Jordi Cortadella, Kyller Gorgônio
Department of Software
Universitat Politècnica de Catalunya, Spain
http://www.lsi.upc.edu

Fei Xia, Alex Yakovlev
School of Electrical, Electronic and
Computer Engineering
Univ. of Newcastle upon Tyne, UK
http://www.ncl.ac.uk/eece

Abstract

Asynchronous data communication mechanisms (ACMs) have been extensively studied as data connectors between independently timed processes in digital systems. In previous work, systematic ACM synthesis methods have been proposed. In this paper, we advance this work by developing algorithms and software tools which automate the major part of the ACM synthesis process. Firstly, an interleaving specification is constructed in the form of a state graph, and secondly, a Petri net model of an "ACM-type" is derived using the notion of an ACM-region. The method is applied to a number of "standard" writing and reading policies of ACMs with shared memory and unidirectional control variables.

1. Introduction

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 [4, 8]. Secondly, concurrent and distributed system implementations lead to greater asynchrony between components as semiconductor technology advances and the degree of integration increases (ITRS-03 "Design" document emphasizes multiple clock domains and source-synchronous signalling and predicts networks of self-timed blocks [1]). The size of computation networks is becoming larger, and the traffic between the processing elements is increasing. Handling the data communications which make up the traffic, therefore, may determine much of the performance and characteristics of such systems.

One of the most important issues in designing communication schemes between asynchronous processes is that such schemes should allow as much asynchrony as possible after satisfying design requirements on data.

An asynchronous communication mechanism (ACM) is a scheme which manages the transfer of data between two processes not necessarily synchronized for the purpose of data transfer. The provider of data is called the "writer" of the ACM, and the user of data is referred to as its "reader". The ACM is therefore a data connector linking the two processes, the writer and the reader. The general scheme of these kinds of data communication mechanisms is shown in Figure 1. Most ACM implementations tend to include shared memory, accessible to both writer and reader, for the data being transferred, and control variables, each of which is usually 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 the writer and reader processes are single-thread loops, during each cycle of which a single item of data is transferred to or from the ACM.

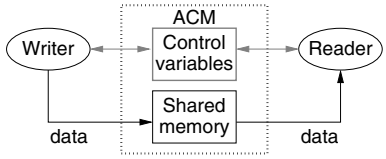


Figure 1. ACM with shared memory and possibly control variables

Classical semaphores can be easily configured to protect write and read operations. However, this can be done only at the large granularity level of data accessing, which is not satisfactory because we expect a minimum locking between the reader and writer processes. This desire is exemplified by the original work of Lamport on "atomic registers" [6] and present in all subsequent research on ACMs (which include the Lamport atomic register) in the literature. One way to achieve this is to design the ACM in a such way that the atomic actions of each process only occurs at a very small granularity level, when accessing control variables, i.e. we need to move the atomicity of actions from data accesses to few bits control variable accesses [8, 11]. Addi-

Proceedings of the Fifth International Conference on Application of Concurrency to System Design (ACSD'05)
1550-4808/05 $ 20.00 **IEEE**

tionally we also expect the ACMs, in certain cases, to avoid busy waiting (by testing an empty/full flag and leaving), and this leads us to multi-slot mechanisms. These are the properties of an ACM that should be reflected in its initial specification.

In this work, a state graph specification is obtained from a functional specification, which can capture the above mentioned properties of an ACM at the level of interleaving semantics, and then its Petri net is synthesized. This contradicts the usual way of synthesizing asynchronous circuits, in which a Petri net specification is first obtained, and then its state graph is constructed. We do it in that way 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 implementation than state graphs, and the generation of the implementation from the models is more natural.

In previous work, we succeeded in devising a step-by-step method, based on the theory of regions, of synthesizing ACM algorithms from interleaving state-space specifications [13, 10]. We also found that the automatic synthesis tool Petrify in its shape does not help much in this task [13]. In this work, we have incorporated our regions-based techniques into a new version of Petrify. The case studies in this paper demonstrate that the modified Petrify does help derive ACM algorithms from interleaving specifications in a reasonably straightforward manner.

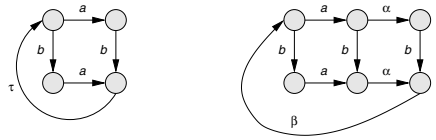
We have also developed an automatic method of generating interleaving state-space specifications from more general functional specifications. Together with the modified version of Petrify, this increases the scope of automation in the process of ACM design and implementation. With these two components, we almost have an entirely automated ACM synthesis apart from the final step of turning state-machine like Petri net descriptions of ACM algorithms (output of the modified Petrify) into software or HDL codes, which will be the next stage of our work.

2. Motivation

2.1. Communication without process sharing

Interprocess data transmission can be implemented traditionally through synchronization for the purpose of data transfer (e.g. Occam [5]). 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 specifications. In previous work, much effort has been spent on making ACMs as asynchronous as possible, given a particular specification. This work continues to aim for this goal.

Functional specifications of ACMs may demand synchronization between the processes at some point. For example, let us consider a simple ACM, called "rendezvous" in [8]. This ACM has two processes with one private action each, a and b, respectively. Every time Process 1 executes action a it must wait until Process 2 has executed action b once, before repeating its action a, and vice versa, as the system is symmetric. This informal functional specification can be represented by an interleaving state graph model shown in Figure 2(a). The state graph contains a special shared action τ, where the synchronization of the two processes happens.



(a) Initial interleaving model (b) Refined model

Figure 2. A rendezvous ACM

This most basic form of synchronization via process sharing involving two otherwise independent processes can be modelled by the Petri net in Figure 3, which can be implemented with a C-element in hardware. This involves an action (the synchronization transition) which belongs to both processes. When one process is ready to carry out this action, it must wait for the other process to reach the same point before both carry out the action together.

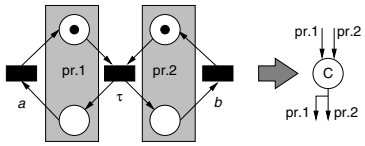


Figure 3. Implementing synchronization with a C-element

Previous work on ACMs has shown that it is possible to reduce the need for this kind of synchronization to the most basic actions in digital systems, i.e. those that can be regarded as atomic, regardless of the functional specification of the ACM [8]–[10]. 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 Figure 2(a) can be refined into Figure 2(b), where the two processes do not share an action (process sharing is removed by replacing τ with actions α for Process 1 and β

Process 1	Process 2
wait until $y = 0$;	wait until $x = 0$;
$x := 0$;	$y := 1$;
do a;	do b;
wait until $y = 1$;	wait until $x = 1$;
$x := 1$;	$y := 0$;

Table 1. Rendezvous algorithm

for Process 2) without losing the functional aspect of the synchronization required. After further refinement by additional synchronization actions, this can be implemented by the Petri net model in Figure 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 3. Algorithm 1 can be derived (for possible software implementation) from the Petri net model in Figure 4, where the two pairs of complementary places can be encoded with variables x in Process 1 and y in Process 2. It is not on the scope of this paper to discuss about a systematic way to derive such algorithm, but we expect to report it in the future.

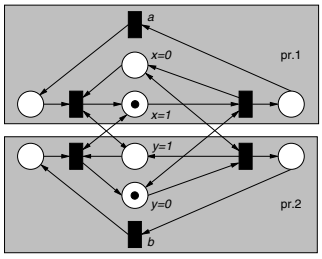


Figure 4. Implementing synchronization with unidirectional control variables

This simple example also illustrates the steps of our ACM synthesis process included in this paper. 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.

2.2. Unidirectional control variables

Furthermore, previous work has also demonstrated that Figure 4 effectively specifies "unidirectional control variables". The value of such a shared variable 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 4 are such variables.

By using unidirectional control variables, any need of synchronization (either demanded by the functional specification or out of implementation necessity) can be removed from non-atomic actions (such as the reading and writing of multi-bit data) to finest grain 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 by Figure 4, all non-atomic actions (assembled into transitions a and b) are fully asynchronous between the two processes. This provides for maximum practical asynchrony for any functional specification, and the safest solution. 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 or ternary) is associated with metastability. 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 it will eventually "resolve" to either 0 or 1 non-deterministically. This is shown in Figure 5.

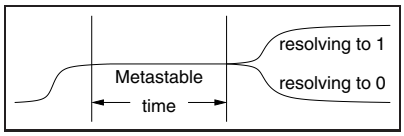


Figure 5. Sketch of metastability in a variable

In practice, the effects of such metastability, which in modern semiconductor technologies (e.g. CMOS) does not include any oscillatory behavior, can be minimized. 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. In fundamental mode operations, 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 this work, we will restrict any synchronization between the reader and writer processes to such unidirectional control variables.

3. Interleaving specification

We use an example to illustrate the type of interleaving specification we need and our method of deriving it. This is a rereading bounded buffer ACM (called RRBB in [10]) with three data cells in the buffer and one slot per cell. The

basic interleaving state graph of this ACM is shown in Figure 6. This state graph includes only the data reading and writing actions.

In Figure 6, rd_i, $i = 0, 1, 2$, indicate reading access of cell i, and wr_j, $j = 0, 1, 2$, indicate writing access of cell j. The two s_0 nodes denote the same state. This is also true for the s_1 nodes. The entire graph is therefore cyclic. It can be seen that the reader is never forced to wait, rereading when necessary, while the writer will wait when the reader is accessing the cell it is scheduled to access.

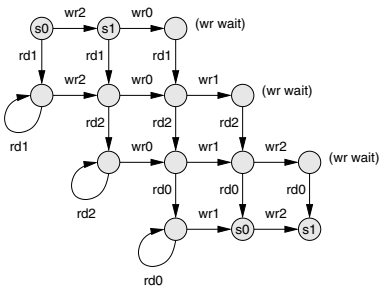


Figure 6. Basic state graph of 3-cell RRBB

This state graph assumes that rereading accesses the newest data item in the ACM.

This type of simple state graph is not suitable for synthesizing ACM algorithms [13, 10]. Both the writer and the reader need to make decisions about whether to wait or whether to re-access the last cell accessed. Such decision making implies "hidden actions" (similar to α and β in Figure 2) not shown in a state graph of the type in Figure 6.

Extending the state graph in Figure 6 to include the necessary hidden actions produced the refined specification in Figure 7. In this figure, λ_{ij} indicates the hidden writer action which advances the writer from cell i to cell j, and μ_{kl} indicates the hidden reader action which either advances the reader from cell k to cell l when $k \neq l$ or prepares for the rereading of cell k when $k = l$.

All data access actions (wr and rd) always form (commutative) diamonds with actions of the other side. This means that these actions are fully concurrent with actions of the other side, maximizing interprocess asynchrony. The hidden actions, however, do not have this property. Therefore all the critical synchronization points in the communication are concentrated on these actions. In order for the resulting ACM to be as asynchronous as possible, it is important that they should take very small amounts of time and be atomic ideally. We assume here that this is accomplished by making these actions the setting and reading of unidirectional control variables of the smallest size.

Previous work has show that when these actions are regarded as non-atomic, not all ACM implementations work according to 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 [9]. As a first effort, we choose to regard control variable actions as atomic.

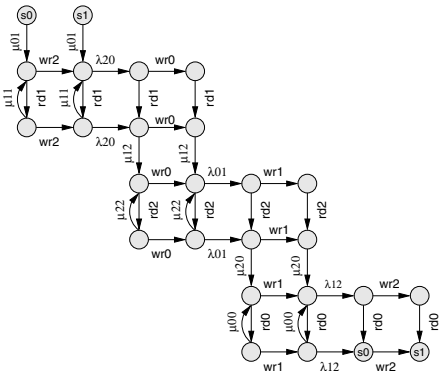


Figure 7. State graph of 3-cell RRBB ACM including hidden actions

Figure 7 indicates that the silent actions of the writer and reader depend on each other. For instance, whether the next reader silent action is μ_{11} or μ_{12} depends on if λ_{20} has completed, and whether λ_{20} may start (or the writer must wait) depends on if μ_{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.

Allowing overwriting brings further problems, and we will illustrate this with a different example here. In the context of ACM, there is usually no assumption about the relative speeds of the writer and the reader and any synchronization between them outside the ACM, and pointing the writer towards the cell to which the reader will next be pointed runs the risk of creating data coherence problems. This problem can be solved through the use of multiple data slots for the data memory of a single cell.

The simplest multi-slot cell consists of two data slots. Figure 8 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 is too big to include here and has 80 states and 160 arcs.

Arcs μ_{ijij} indicate the reader preparing to reread 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. Depending on where the writer is, the reader may advance to the next cell or continue advancing until it reaches the correct cell. μ_{0111} and μ_{0100} branch off to different parts of the state graph, and although μ_{0100} appears to be the reader staying in the same cell, it is actually the reader advancing to the next cell, finding the writing accessing it, and then advancing back to cell 0 again.

Evidently, as the size of the ACM increases, or more complex policies are required, it becomes more complicated

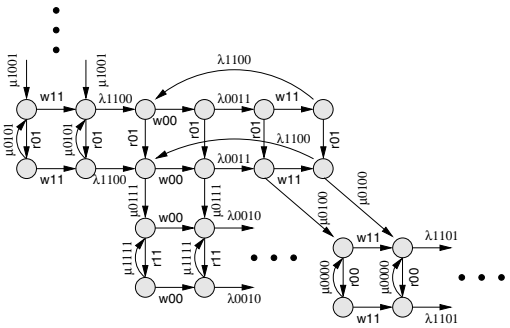


Figure 8. Partial state graph of 2x2-cell OWRRBB ACM including hidden actions

for a human to manage the complexity of specifying the behavior of the processes through a state graph.

4. Deriving the state graph specification

One of the purposes of our work is to provide the designer of asynchronous systems with an automatic procedure to generate the state graph specification, that satisfies the interleaving properties presented in Section 3, of an ACM given the function that should be implemented by the ACM and its size. The designer should not need to manage the complexity of specifying how blocking of data access (read or write) is avoided and how data coherence is guaranteed during the execution of the system.

4.1. Supported ACM function types

The generated state graph specification should preserve some properties that are inherent to the function implemented by an ACM. At the moment we are able to generate three different types of ACM functions:

1. **RRBB**: rereading is allowed but not overwriting.
2. **OWBB**: overwriting is allowed but not rereading
3. **OWRRBB**: overwriting and rereading are allowed

We classified the ACMs according to whether overwriting and rereading are permitted [14]. For rereading, it is much more natural to reread the item read in the previous cycle rather than reread an item read several cycles before. On the other hand, overwriting might happen anywhere in the buffer. However, the strongest practical cases have been proposed for overwriting either the newest or the oldest item in the buffer [8, 12, 3]. Overwriting the newest item in the buffer, which is is a relatively straightforward task to approximate [12], attempts to provide the reader with the best continuity of data items for its next read, data item continuity being 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, which is true for many types of applications. Here we attempt to tackle the much more interesting problem of overwriting the oldest item in the buffer, to which the reader will naturally be pointed next.

Therefore, as stated in Section 3, allowing overwriting necessitates two slots per cell, whilst not allowing it needs only one slot per cell.

4.2. The synthesis procedure

The important part in the synthesis procedure is 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 example, in the RRBB type 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 types that allow overwriting it is necessary to have this information. Here we will explain in detail how to calculate a new state for the RRBB type. The same scheme is used in the OWBB and OWRRBB policies, but with extra variables.

Here is a formal definition of an ACM state graph specification.

Definition 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 of the ACM;*
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.*

We say that a state s_n is 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 $n > 0$, i.e. there exists a sequence of actions that drives from s_0 to s_n.

For an ACM that permits only rereading, each state of the ACM is defined in terms of the values of the variables that controls which cell each process will access, or is accessing, data. Since we are taking into account that a process may be ready to access or accessing data in a cell i it is necessary to distinguish between these two internal states of the process. So, a state in the ACM will be determined by four variables, two that denote the cell that the processes are pointing to and two that specify if a process is accessing the buffer or ready to access it. Definition 2 formally presents this concept.

Definition 2 (State) *A state s of a RRBB ACM is a vector $[i, j, k, l]$, with $i, j \in \mathcal{N}$ and $k, l = 0, 1$, where:*

1. *i determines the cell number the writer is pointing to;*
2. *j determines the cell number the reader is pointing to;*
3. *k determines if the writer is ready to access ($k = 1$) data in the buffer or is accessing ($k = 0$) the buffer;*
4. *l determines if the reader is ready to access ($l = 1$) data in the buffer or is accessing ($l = 0$) the buffer.*

s_{ijkl} will be used as a label to state s when we desire to show the status of the variables of s explicitly. For example, if a state has the label s_{0110} it means that the writer process is ready to access cell number 0, and next action of the writer is to write data in the cell. And the reader is accessing data in cell 1 and will next carry out a hidden action.

The transition relation is determined by a set of rules that, when correctly applied, will *"control"* the behavior of the ACM in a way that the resulting behavior will corresponds to some specific function type. Definition 3 captures the formal concepts of the transition rules for RRBB ACMs. We use $a \oplus 1$ do denote $(a + 1) \mod n$.

Definition 3 (Transition rules for RRBB ACMs) *A transition t is a valid RRBB ACM transition if one of the following applies:*

1. *If $s_{ijkl} \longrightarrow s_{i'jk'l}$ then:*
   - *(a) $k = 1 \Rightarrow i' = i \wedge k' = 0$*
   - *(b) $k = 0 \wedge j \neq i \oplus 1 \Rightarrow i' = i \oplus 1 \wedge k' = 1$*
   - *(c) $k = 0 \wedge j = i \oplus 1$, then the transition cannot be executed, and the writer waits until the reader advances to another cell.*
2. *If $s_{ijkl} \longrightarrow s_{ij'kl'}$ then:*
   - *(a) $l = 1 \Rightarrow j' = j \wedge l' = 0$*
   - *(b) $l = 0 \wedge i \neq j \oplus 1 \Rightarrow j' = j \oplus 1 \wedge l' = 1$*
   - *(c) $l = 0 \wedge i = j \oplus 1 \Rightarrow j' = j \wedge l' = 1$*

   where n is the number of cells in the ACM.

Condition 1(a) tells us that if the writer process is ready to access ($k = 1$) the memory cell i, then the next action will be to write data in the cell. On the other hand, condition 1(b) says that if the next action of the writer is a hidden action λ (i.e. the process is writing data in the buffer, or has finished doing it) and the reader is not pointing to the next cell, then the writer will point to the next cell with λ. If none of these conditions is valid, then the writer could not execute and it will block until the reader executes.

In a similar way, condition 2(a) tells that if the reader is ready to perform data access ($l = 1$) on cell j, then the next action is a read from the cell. Condition 2(b) tells that if the reader is going to carry out its hidden action μ and the writer is not pointing to the next memory cell, then the reader will point to such a cell with μ. And finally, condition 2(c) says that if the reader is going to update its control variable and the writer is pointing to the next memory cell, then the reader will prepare to reread the data in cell j.

For function types that allow overwriting it is necessary to have four extra variables to identify the states. Two to identify the data slot to be accessed by a process, 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.

In Figure 9 we show part of the state graph generated for a 3-cell RRBB ACM using our new algorithmic method. The entire state graph is isomorphic to the one in Figure 7. The initial state is labelled with **2010** (writer ready to access the data slot, reader ready to perform a silent action).

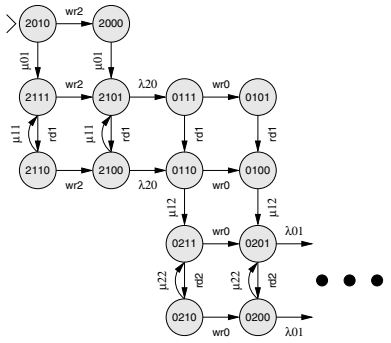


Figure 9. Generated state graph of 3-cell RRBB

The generation of the successors of the initial state is done by applying the rules 1(a) (state **2000**) and 2(b) (state **2111**). The next step is to generate the successors of such states. Since, in state **2000**, the execution of the writer satisfies neither 1(a) or 1(b) the writer cannot be executed, and the execution of the reader reaches state **2101** by applying rule 2(b). The application of rules 1(a) and 2(a) in state **2111** leads to states **2101** and **2110**. The execution of the steps above for all states will generate the entire state graph of the ACM.

Using the guidelines above we implemented a tool, called Jabuti, that can generate a state graph specification for RRBB, OWBB and OWRRBB ACMs that satisfies the interleaving specification defined in Section 3.

5. Petri nets synthesis methodology

We approach here the problem of synthesis of asynchronous communication algorithms as a problem of synthesizing a Petri net of a certain class. 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. Our 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 [7].

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 [2]).

Informally, such synthesis is a decomposition, or distribution, of global states of the state graph into 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, originating from [7], and regions have one-to-one correspondence to places in the synthesized net.

5.1. Regions

A *region* is a subset of states in which all arcs labelled 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 labelled 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-labelled 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 10(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 the region and arcs not crossing the region.

It is known from [2] that, in order to generate a 1-safe Petri net (a net in which places never get more than one token in every reachable marking) whose reachability graph is isomorphic to a given state graph, the state graph must satisfy the important properties of *state* and *state-event separation*.

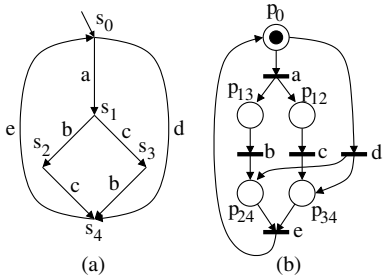


Figure 10. (a) State graph and (b) Petri net.

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 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 labelled with e).

The basic procedure to produce a 1-safe Petri net from a state graph satisfying the above properties is as follows:
1. For each event label 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 (canonical) procedure, if applied, would generate the so-called saturated net [2], 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 [2]. The resulting Petri net reflects the notion of concurrent operation between events forming commutative diamonds in the interleaving (i.e. state graph) form.

Figure 10(b) depicts a Petri net obtained from the synthesis of the state graph in Fig. 10(a). The subindices 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 the 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.

5.2. Synthesis for ACMs

Nets obtained by the aforementioned synthesis methods 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.

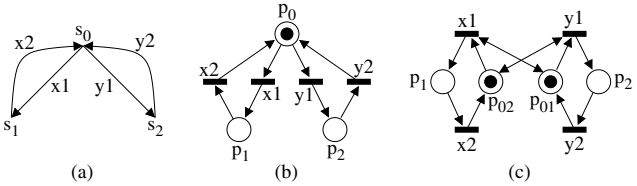


Figure 11. (a) State graph, (b) Petri net, (c)Petri net with ACM-regions.

For the example of the state graph of the RRBB ACM, our goal is to obtain a Petri net that consists of two sub-nets, one containing events that are labelled with wr and λ_{ij} and the other with events labelled with rd and μ_{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 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.

The constraints imposed by ACMs can be illustrated by the example depicted in Fig. 11. Figure 11(b) shows the Petri net obtained from the synthesis of the state graph in Fig. 11(a). 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.

Unfortunately, the net in Fig. 11(b) does not fulfil the requirements for an ACM, since place p_0 cannot be associated to any process (it can be either modified by processes x or y). By using a different set of regions, the net in Fig. 11(c) 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 region called *ACM-region*. 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 [2], using ACM-regions instead of regions. Petrify was modified to include this technique.

6. Case studies

This section shows two examples of the application of the methodology described in the previous sections, starting from the initial specification until the generation of the Petri net modelling the behavior of each process.

The first step is done by the designer of the system, and consists in the specification of the behavior of the processes composing the system. As we state before, we want to provide the designer with the possibility of not making much effort on it. So, instead of specifying the complex composition of the writer and reader processes, the designer only needs to specify the size and the function type implemented by the ACM. The second step corresponds to the synthesis of the state graph specification, and the third step to the generation of the Petri net from such a state graph. The last two steps are performed automatically.

6.1. 3-cell RRBB

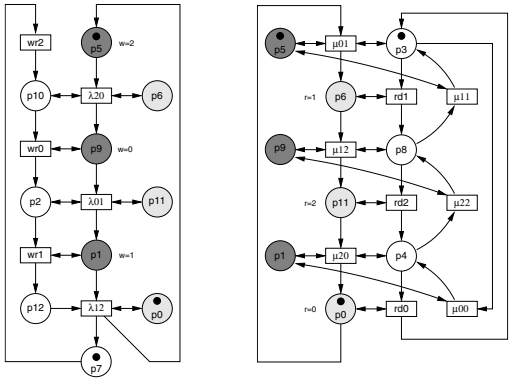
To generate the 3 cell RRBB ACM described in Section 3, it is only necessary to provide a textual input like:

```
channel ACM 3 message;
```

In the example above, the channel declaration on the first line specifies a name (`ACM`), a capacity (`3` memory cells) and a type (`message` meaning RRBB) for the channel that will be used to connect the two processes. The designer is not troubled with deciding such details as how many slots a cell should have, which is determined by the ACM type.

The next step is to transform this specification to a state graph that satisfies the interleaving defined in Section 3, using Jabuti. The resulting specification, which is isomorphic to the state graph of Figure 7, can be processed using the techniques described in Section 5 to obtain a Petri net model that captures the behavior of the reader and writer processes.

Figures 7 and 12 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 if $\lambda 20$ has completed, and whether $\lambda 20$ may start (or the writer must wait) depends on if $\mu 11$ has completed. This means that these silent actions set uni-directional control variables for the other sides silent actions to read. The simplest set of uni-directional control variables included in



(a) Writer process (b) Reader process

Figure 12. Petri net model for a 3-cell RRBB

the μ and λ actions consists of one writer index variable w and one reader index variable r.

Figure 12(a) and 12(b) shows the Petri nets generated by Petrify, with the result from Jabuti as the input, for the writer and reader processes respectively. The places with the same label are the same. A token in p5 means $w = 2$, in the same way a token in p6 means $r = 1$.

In the initial state the writer is ready to write on cell 2 (transition $wr2$ is 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 reader is accessing, or will next access, the next cell the writer is preparing to access, 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 into 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 reread.

Both choices, between two possible μ actions and between a λ action and writer waiting, are determined by how the current values of w and r compare with each other. For instance, if w is set to the current value of r, writer must wait until reader has changed r before proceeding. Following this line of reasoning, an algorithm for the RRBB ACM may be derived (see Table 2). The control variables w and r are initialized with values 2 and 0 respectively [10].

6.2. 4-cell OWBB

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

Writer	**Reader**
write cell w;	if $(r+1 \mod 3) \neq w$ then
$w := (w+1 \mod 3)$;	$r := (r+1 \mod 3)$;
wait until $r \neq w$;	read cell r;

Table 2. 3-cell RRBB ACM algorithm

Figure 13 presents part of the generated state graph for a 4-cell RRBB ACM, with the initial state in node 0. The entire graph has 928 states and 1824 arcs, and it is too big to be shown here.

As in Figure 8 the overwriting cycles can be identified, e.g. $w00 \longrightarrow \lambda 3001 \longrightarrow w01 \longrightarrow \lambda 0111 \longrightarrow w11 \longrightarrow \lambda 1121 \longrightarrow w21 \longrightarrow \lambda 2130$. Depending on where the writer is, the reader may advance to the next cell, continue advancing until it reaches the correct cell or wait until there is new data in the ACM. Thus, the execution of a silent action by the reader branches off to different parts of the state graph that follow the same pattern. Each pair of black dots linked by a dashed arc represents the beginning of one of these branches. Note that the occurrence of the same action, e.g. $\mu 3101$, branches to different subgraphs. It happens because it is necessary to avoid the reader and writer accessing the same cell and slot pair.

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 two big to be included here.

7. Conclusions and future work

We have automated two key steps in ACM synthesis. These are the deriving of an interleaving state graph specification given a functional specification, and the generation of an ACM Petri net model in the form of independent state machines using unidirectional shared variables. Previously, these steps were the most tedious and time-consuming tasks in the manual synthesis process and although suggested to be "automatable" because of their systematic and step-by-step nature, 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. A library of Petri net models of such buffers can be targeted to subsequent hardware or software implementations.

We believe our results may have implications outside the immediate area of ACMs. The modified Petrify, for instance, may be useful in dealing with any systems involv-

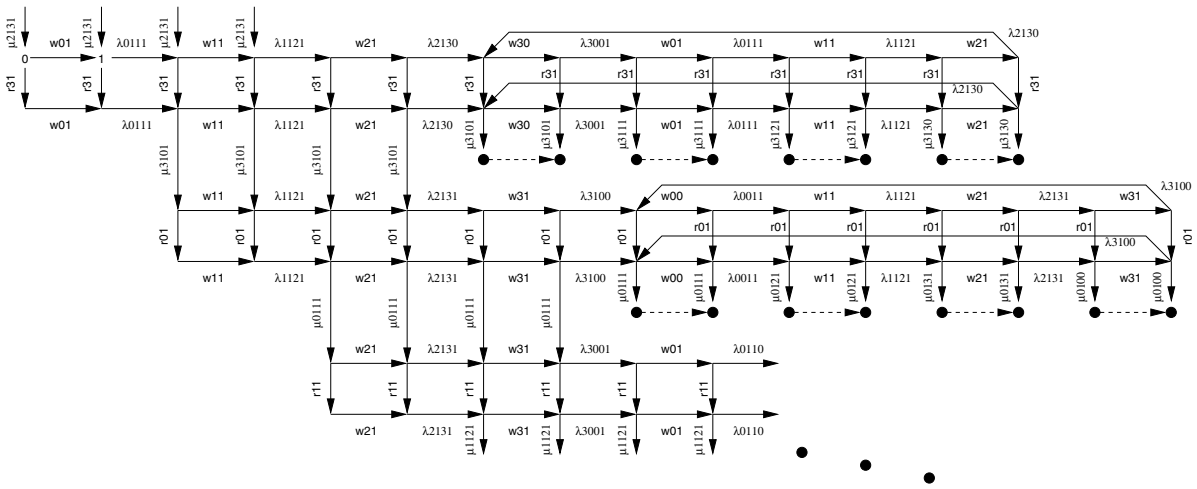


Figure 13. A 4-cell OWBB ACM specification

ing independent processes communicating with each other through unidirectional shared variables. Because using such variables maximizes asynchrony among processes this may be significant in future VLSI systems, in particular globally asynchronous locally synchronous (GALS) systems.

Our next task is the automatic derivation of ACM algorithms (in both software and HDL forms) from the ACM Petri net implementations obtained from the current process. With that, a completely automated ACM synthesis flow will become available for system designers who are not necessarily familiar with the internal details of ACMs.

Acknowledgements. This work is part of the Coherent project (http://async.org.uk/coherent) and supported by grants EPSRC GR/R32666 at the Newcastle University and CICYT TIN2004-07925 at the Technical University of Catalunya. The authors benefited from extensive discussions with I.G. Clark, H. Simpson and E. Campbell and wish to express our gratitude.

References

[1] ITRS: http://public.itrs.net/files/2003itrs/home2003.htm.

[2] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Deriving Petri nets from finite transition systems. *IEEE Transactions on Computers*, 47(8):859–882, Aug. 1998.

[3] J.-P. Fassino. *THINK: vers une architecture de systèmes flexibles*. PhD thesis, Ecole Nationale Supérieure des Télécommunications, Dec. 2001.

[4] C. K. IV, V. Ekanayake, and R. 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.

[5] G. Jones and M. Goldsmith. *Programming in occam 2*. Prentice Hall International, 1988.

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

[7] M. Nielsen, G. Rozenberg, and P. S. Thiagarajan. Elementary transition systems. In *Selected papers of the Second Workshop on Concurrency and compositionality*, pages 3–33. Elsevier Science Publishers Ltd., 1992.

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

[9] F. Xia and I. 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.

[10] F. Xia, F. Hao, I. Clark, A. Yakovlev, and G. 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.

[11] F. Xia, A. Yakovlev, I. G. Clark, and D. Shang. Data communication in systems with heterogeneous timing. *IEEE Micro*, 22(6):58–69, 2002.

[12] A. Yakovlev, D. Kinniment, F. Xia, and A. Koelmans. A fifo buffer with non-blocking interface. *TCVLSI Technical Bulletin*, pages 11–14, Fall 1998.

[13] A. Yakovlev and F. Xia. Towards synthesis of asynchronous communication algorithms. In B. Caillaud, P. Darondean, L. Lavagno, and X. Xie, editors, *Synthesis and Control of Discrete Event Systems. Part I: Decentralized Systems & Control*, pages 53–75. Kluwer Academic Publishers, Boston, Jan. 2002.

[14] A. Yakovlev, F. Xia, and D. 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*, page 127. IEEE Computer Society, 2001.