Towards synthesis of asynchronous communication algorithms

A. Yakovlev and F. Xia*

Abstract

This paper presents preliminary ideas about automated synthesis of asynchronous communication mechanisms (ACMs). Such ACMs can be used in systems with heterogeneous timing, e.g. embedded real-time control and command systems. The problem of constructing communication structures for such systems is known to be hard, and up to now was typically solved on an ad-hoc basis, using an assemble-and-verify approach. An alternative step-by-step method of synthesizing ACM algorithms from state space specifications is demonstrated in this paper via the process of deriving a distributed concurrent algorithmic implementation for an ACM, called Signal, which can be used to interface a data-driven ("lazy") subsystem to a time-driven ("busy") environment.

1 Introduction

In heterogeneously timed systems, data interfaces often need to be maintained between subsystems not belonging to the same timing domain. The minimal form of this problem is the unidirectional passing of data between two single-thread processes.

When the two communicating processes are not synchronised, it is often necessary to pass the data through some intermediate data repository, usually in the form of shared memory. This is schematically shown in Figure 1.

Figure 1: Passing data via shared memory.

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

Lamport classified ACMs into "safe", "regular" and "atomic" "registers" [1] and tried to assemble the more useful regular and atomic types out of directly realizable safe types. Exploiting the fact that the smallest such "registers" (ACMs) that transmit bit type data items can be implemented to atomic standards directly, many ACM protocols have been proposed in the literature (e.g. [6, 2, 3]) which employ less than safe registers for the data path whose access by the reader and writer processes is managed by minimal (bit) control variables. The results of these efforts are much better and much more practical ACMs performing to atomic or regular standards. All these ACMs allow full temporal independence between the reader and writer processes.

Simpson then proposed a more general classification system for ACMs, incorporating protocols that do not necessarily allow full temporal independence for reader and writer [4, 5].

We further developed this classification system by concentrating on the asynchrony characteristics of various ACMs [12], using the more general concept of whether an ACM protocol may or may not oblige waiting for either access process as the differentiating factor. This new system is shown in Table 1. In this table, RW means "reader waits when all data in ACM has been read before", WW means "writer waits when no data in ACM has been read", and RNW and WNW mean "reader never waits" and "writer never waits" respectively. In this system an ACM has a capacity, which is the number of different data items it may contain. The basic ACM data state is the number of unread data items it contains, which varies from 0 to its capacity. The best methods with which reader and writer can avoid waiting are re-reading (if data state is 0) and overwriting (if data state equals the ACM's capacity), respectively.

Constructing algorithmic descriptions for such ACMs that could be easily translated either to hardware or software implementation has proved to be an intellectually challenging task. For example, the four-slot algorithms for the Pool ACM, developed by Simpson [6], were non-trivial solutions, which required developing validation techniques that were

*The authors are with the Department of Computing Science, University of Newcastle upon Tyne, Newcastle upon Tyne, NE1 7RU, UK

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

	RW	RNW
WW	Channel	Message
WNW	Signal	Pool

quite complex and hard to check on their own [7]. Apart from recent attempts by the authors of this paper, all other published ACM solutions have been obtained via a process of intuition-based ad hoc design. Although there has been at least one previous attempt at finding a method to automatically synthesize point to point data communication interfaces [9], it only works when both processes are controlled by the same clock, i.e. fully synchronized.

In this paper we demonstrate the process of deriving a distributed concurrent algorithmic implementation for an ACM of the Signal type, which can be used to interface a data-driven ("lazy") subsystem to a time-driven ("busy") environment. The synthesis flow proceeds through the stages of: (i) constructing a State Graph specification of the Signal, (ii) refining it with silent actions which are subsequently interpretted as actions that change the values of control variables, (iii) synthesizing a Petri net which decomposes the system into two sequential processes that interact by reading each other states by means of test or read arcs, and finally (iv) mapping the Petri net system into an algorithmic description where the processes use guarded and unguarded assignments statements defined on control variables. A hardware implementation of a two-slot Signal ACM using self-timed circuits was described in [12].

This work is only a preliminary study in the direction of automated synthesis of communication structures. In the future we plan an indepth study into the key steps of the above synthesis procedure, developing a more formal framework. For example, appropriate constraints for state and state-event separation, within theory of regions [13], could be defined to support step (iii). More work on formalising the refinement with silent actions, particularly those with meaningful semantic interpretation, would be needed for step (ii). Furthermore, the use of state graphs as an initial specification language is not necessarily the best option. Temporal logic based techniques [10, 11] may merit further investigation.

2 Definition of the basic Signal

The basic Signal protocol specifies an ACM with capacity 1, re-reading not permitted and overwriting permitted. In other words, writing can happen when the Signal contains either 0 or 1 item of unread data. A write data access modifies the data state of the Signal to 1, but reading can only happen when there is one item of unread data in the Signal.

This is a special case of the general definition of the Signal with capacity n, $n \geq 0$, of which the "overwriting buffer" schemes found in many places in the literature (such as [19]) are also special cases.

This basic Signal definition is captured by the Petri net model in Figure 2. Here the places "0" and "1" are complementary to each other, one and only one being marked at all times.

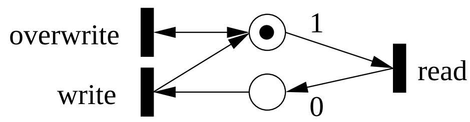

Figure 2: Basic definition of the Signal protocol.

This definition treats the write and read data accesses as atomic processes, which is not sufficiently clear for system synthesis and implementation. In reality, data accesses by the writer and reader must take time and the timing relationship between the reader and writer processes is important for an ACM in a real-time system.

In other words, apart from the timing requirements imposed on the reader and writer by the data state of the ACM, data accessing of one side may affect the temporal behaviour of the other side. In other words, an implementation may or may not require one side to wait when the other is in the middle of a data access. The model in Figure 2 is not specific enough about such distinctions.

By using techniques introduced in [18], a refinement model is shown in Figure 3. By treating the read and write data accesses asymmetrically, this definition maintains the possibility of full temporal independence for the writer, but prescribes waiting for the reader while either writing or overwriting is in progress. It also means that the reader will always obtain the newest item of data from the writer available at the time of reading and the writer is always allowed to access the Signal, regardless of the data state of the Signal or the state of the reader. This definition is used in this paper. Note that the place "nw" denotes the situation when there is no write access taking place. Only when this place is marked can the reader initiate an access.

Formally, the definition in Figure 3 reflects the following properties expected from the ACM:

1.Data states and their updating:

The Signal has a data capacity of 1. In other words, at any time, it has either 0, 1 or undefined (nondeterministic) item of unread data.

At the start of a read data access, the Signal's data state is set to 0 (unread := 0 and read := 1).

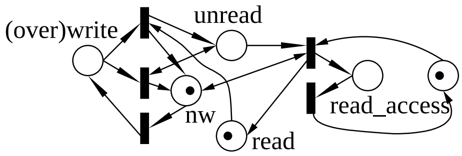

Figure 3: Signal with non-atomic writing.

At the start of a write data access, the Signal's data state is set to undefined (nw := 0).

At the end of a write data access, the Signal's data state is set to 1 (unread :=1 and read := 0), the item of data provided by this write data access being unread.

2. Conditional asynchrony for the reader:

The reader can start data access only when the data state of the Signal is 1 but a read data access can be arbitrarily long.

3. Unconditional asynchrony for the writer:

The writer must be allowed to start and complete a data access at any time, regardless of the data state of the Signal and the status of the reader.

4. Data coherence [6]:

The Signal and the data accesses of the writer and reader processes must not modify the content of any item of data. In other words, any item of data received by the reader must not have been changed since being provided by the writer.

5. Data freshness [6]:

Any read data access must obtain the data item designated as the current unread item in the Signal, i.e. the item of data made available by the latest completed writer access.

In the terminology of multi-slot ACMs [6], a "data slot" is a unique portion of the shared memory which may contain one item of data. It is obvious that a Signal in the form of Figure 3 cannot be implemented with only one data slot, since it cannot possibly support writing and reading at the same time and maintain data coherence. In other words, properties 3 and 4 cannot both be satisfied by an implementation with only one data slot.

Previous work has indicated that it is desirable to minimise the number of slots in multi-slot ACM implementations [14, 6]. The advantages include smaller hardware expenditure both in the actual slot memory areas and control circuits, leading to better temporal performance and higher reliability.

Most of the software solutions in the literature also spend considerable effort in the reduction of the number of slots needed for any particular ACM specification, with similar reasons.

It is therefore important that a solution with just two data slots be found if possible.

3 State graph specification of a simple Signal

The conceptual definition of Signal in Figure 3 cannot be used as a formal specification of the Signal protocol because it does not show the specifics of this protocol, such as how blocking on writing is avoided by using mutiple slots. In this section we construct a state graph specification for a two-slot Signal, which will define a maximally permissible automaton satisfying the required properties of the ACM (see Figure 4). We first formulate these properties using the notion of states and transitions available in state graphs. State graph modelling provides an intuitively easy specification language due to the use of global states and interleaving (sequential) semantics. A Petri net would already be a decomposition of the system's states into local states (places) and this is often non-trivial when the system consists of a number of processes (write and read) and components (data slots).

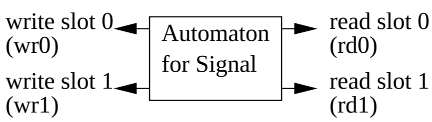

Figure 4: Automaton of Signal.

Actions: Actions are processes whose start and completion are atomic events and whose durations are finite but non-atomic.

States: A state is the result of the completion of one or more actions. During a state, there may exist actions which have started but not completed (in process) and there may exist actions which may start. These "in process" and "may start" actions cannot complete, however, without resulting in a new state.

Previous sets: The previous set of a state s, denoted as P_s, is the set of actions whose completion resulted in s.

Next sets: The next set of a state s, denoted as N_s, is the combined set of actions which may start during s and actions which may be in process during s.

In this paper, the slots will be known as slot (0) and slot (1); write data access to slot (i) is known as wr$i$ where $i = 0, 1$; read data access to slot (j) is known as rd$j$ where $j = 0, 1$. These are the only actions considered at the moment. So during each write cycle the writer performs one action wr$i$ with some i and during each read cycle the reader performs one action rd$j$ with some j. With these assumptions, properties outlined in the previous section require the following conditions:

1. Data states and their updating, and asynchrony for reader and writer:

$\forall s$, i and j: wr$j \notin P_s \Rightarrow$rd$i \notin N_s$; this means that if a state was not the result of a completion of a write data access (i.e., it is solely the result of the completion of a read data access), then a new read data access cannot start during it (data state = 0).

$\forall s$: $\exists i$: wr$i \in N_s$; this means that a write data access must be allowed to start or be in process during any state.

2. Data coherence:

$\forall s, i, j$ and $i = j$: $\neg$(wr$i \in N_s$&rd$j \in N_s$); this means that there can be no simultaneous read and write access of any slot.

3. Data freshness:

$\forall s$ and $i \neq j$: wr$i \in P_s$&rd$j \in P_s \Rightarrow$ wr$j \in N_s$&rd$i \in N_s$; this "slot swapping" fully utilises the two available slots so that the reader always obtains the item of data provided by the latest completed write data access.

4. No "retry loops":

rd$j \in N_{s_m}$ &rd$j \notin P_{s_k} \Rightarrow$rd$i \notin N_s$, $\forall s_m$ preceding s, $i \neq j$, and for all s_k on the state trajectory between s_m and s, including s_m; this is also true for write data accesses.

Figure 5 illustrates these constraints.

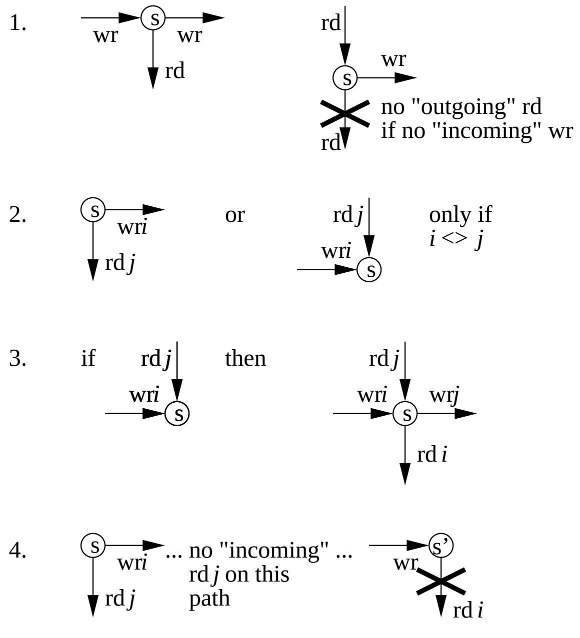

Figure 5: State graph constraints for the two-slot Signal.

Condition 4 reflects the desire to avoid "retry loops" [8], so as to keep the solution simple. It means that, once the reader (or the writer) has been allocated a slot for access, it must perform this access before it can be allocated the other slot for access.

A simple state graph specification has been obtained from these conditions. It conforms to the definition in Figure 3 and is shown in Figure 6.

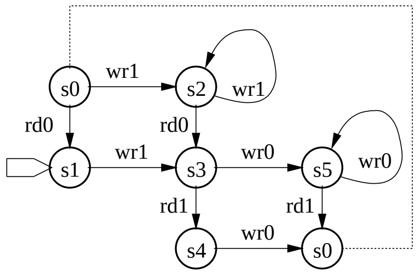

Figure 6: Simple state graph specification for a two-slot Signal (s1 is the initial state).

In Figure 6, the dotted edge denotes that the two states at its ends are essentially the same state and the state graph is in open form for readability.

All conditions mentioned above have been incorporated into the specification. For instance, at state s2 in Figure 6, rd1 cannot be in the next set because rd0 was in the next set at state s0 and has not featured in the previous set of any state on the trajectory from s0 to s2 (Condition 4). At state s1, no reader slot access is in the next set because there is no writer action in its previous set (Condition 1).

This specification is not detailed enough for implementation, because the actual means which maintains the slot steering for the writer and reader processes remain undefined. A further refinement is obtained by adding "silent actions", which perform the necessary functions of the slot-management control variable statements used in published multi-slot ACMs [6, 14]. The result of this refinement is shown in Figure 7. This refinement will also be needed in order to satisfy the separation conditions for Petri net synthesis, described in the following section.

The result of this "refinement", however, is not completely equivalent to the original (e.g., the original state s3 does not have a wr1 self loop arc attached). This is because the original output events (wr's and rd's) must be kept persistent. Therefore conflict resolution must be at the level of silent actions. In the practical sense, whilst the completion of rd0 from s2 should result in s3 instantaneously in the state graph in Figure 6, the refinement recognizes that the recording/reporting of the completion of rd0 is not possible without the help of a reader silent action. So before the completion of this silent action, the writer's slot should not change from 1 to 0.

Both halves of Figure 7 show the same state graph. The reason for duplication is simply to avoid cluttering when deriving regions corresponding to the actions related to write and read parts. In this figure, the λ's denote silent actions that separate the states with the same connections to the main actions of data slot accessing. For instance, $\lambda 0$ is the control action that prepares the Signal for giving next access to slot

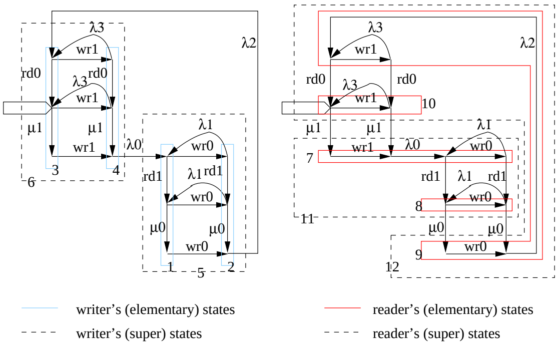

Figure 7: "Distributing" the global states between reader and writer parts using regions.

1 to the reader and next access to slot 0 to the writer. Before $\lambda 0$, writer was allocated slot 1 and reader was allocated slot 0. The μ's denote silent actions performed in the read part. For example, $\mu 1$ stands for the request of the reader to start reading slot 1, and also the fact that slot 0 is no longer used for reading. This should indicate to the writer that it should move to slot 0 if it has finished or when it finishes with slot 1. Hence, depending on whether action $\mu 1$ has or has not been performed by the reader, the writer decides whether to do $\lambda 0$ (move to writing slot 0) or to do $\lambda 3$ (keep writing slot 1).

4 Petri net synthesis

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 [16]). 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 data accessthe Petri net. More formally, synthesis is based on the concept of *regions* in transition systems, originating from [15], and regions have one-to-one correspondence to places in the synthesised net. 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 subset of states labelled with 6 in Figure 7 is a post-region for event $\lambda 2$, pre-region for event $\lambda 0$ and co-region for events wr1, $\lambda 3$ and rd0.

It is known from [16] 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.* Informally, the state separation property requires that for any two different states there must exist a region which contains one of the states and does not contain the other. The state-event separation (also known as forward closure) 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 satifying 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 bidirectional 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 [16], 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 synthesised 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 [20]. The result of such synthesis is shown in Figure 8.

This Petri net reflects the notion of concurrent operation between write and read parts at the level of its primary events wr and rd, as well as its places. The question however arises, does this net completely satisfy our intuitive requirement (about the algorithmic implementation of Signal) of *constructing a pair of sequential processes that must interact by reading each other's local states?*

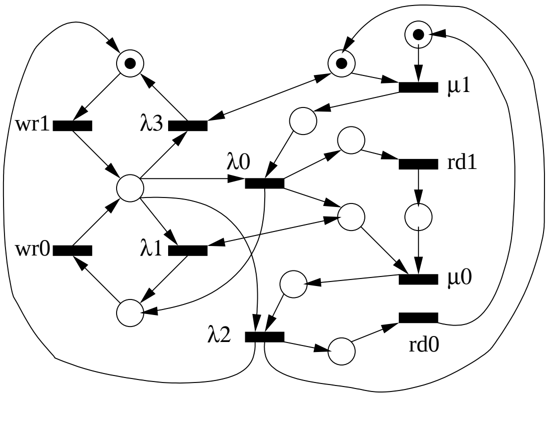

Figure 8: Petri net specification from Petrify.

Secondly, would it be possible to formulate this requirement as a constraint to the Petri net synthesis procedure? More precisely, could we refine the state and state-event separation condition in such a way that only permitted region-event pairs were used to generate the original state graph behaviour?

Let us examine this net from the point of view of its subsequent mapping onto an algorithmic description which is constrained by the above intuitive statement. All transitions labelled by the rd and μ names belong to the read part process. In the write part, however, only some events are properly located, namely those labelled with wr and $\lambda 1$ and $\lambda 3$, whereas $\lambda 0$ and $\lambda 2$ are shared between the processes. These transitions are connected to the net places belonging to both processes by consuming arcs and thus, both processes change their states when they fire.

For correct distribution of the Petri net we will require the following *distribution condition* to be satisfied. Firstly, the net, N, must be decomposable into two subnets, the write subnet N_w and the read subnet N_r, which are (sequential) state machines at the level of their minimal regions[1]. Note that each of these nets need not be an FSM net [17] as a whole because some of it's places may be marked concurrently with other places; it is important that such concurrently marked places are not minimal regions and that they are used to represent the state of control variables which may be tested by the counterpart net. Secondly, N_w and N_r are connected to each other *only* by means of read (testing) arcs. More formally, for every transition t belonging to one of the sub-nets, its pre-places p which belong to the other subnet must always be connected to t by read arcs, i.e. be co-regions to t at the state graph level.

Fortunately, it is possible to find a set of regions to satisfy both state and state-event separation axioms and the distribution condition. A distributable Petri net has been produced using the regions shown in Figure 7. This net is shown in Figure 9. Its reachability graph is isomorphic to the state graph specification. This net is formed by two subnets, with places labelled 1 to 6 belonging to the write subnet and places 7 to 12 belonging to the read subnet. Note that places 5 and 6 in the N_w net and places 11 and 12 in N_r are non-minimal regions. These places provide interface through which the two subnets interact by using bidirectional arcs, which are also known as test or read arcs. Another important issue about these places is that they form complimentary pairs (e.g. when 5 is marked 6 is unmarked and vice versa), which suggest a convenient algorithmic interpretation for them, in terms of binary control variables.

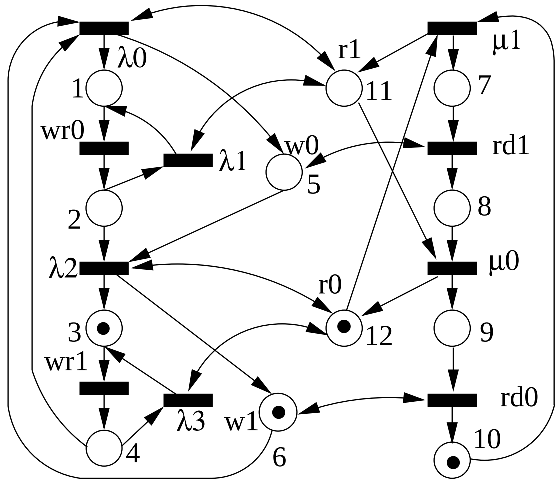

Figure 9: Distributable Petri net for a two-slot Signal

5 Algorithmic Implementation

During synthesis of the distributable net in the previous section we effectively decomposed the entire net system into two sequential processes that communicate through two binary variables, w (write) and r (read), that can be associated with two pairs of complementary places, 5 and 6 for the zero and one states of w, and 11 and 12 for the one and zero values of r. It can also be seen that the λ transitions are part of the write actions and the slot addressing mechanism. They test the r signal and switch the value of the w. Similarly, the read actions are the actions that test the w signal, while also switching the state of the r signal. Thus the labelling of write and read actions can be explicitly associated with the values of the control variables by using indexing. Namely, wr(w) would mean that we write the slot addressed by the value of w. Similarly for rd(r).

To further illustrate the idea of mapping a distributable Petri net into an algorithmic description

[1] Regions that are not unions of other regions.

let us consider the write part of Petri net of Figure 9. It is shown separately in Figure 10.

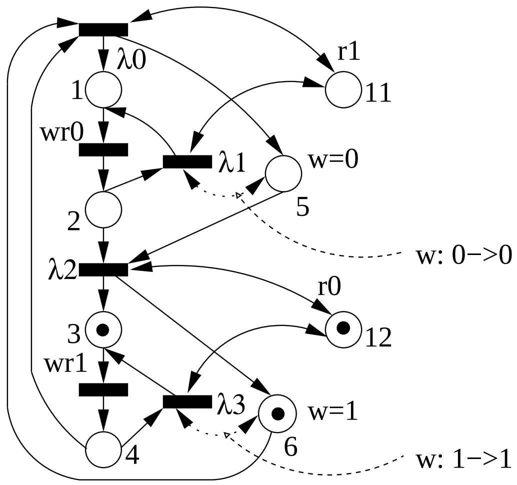

Figure 10: Write part of Figure 9.

The mapping process consists of finding an algorithmic interpretation for both places and transitions. The interpretation of places 1 to 4, which correspond to minimal regions is concerned with the control flow constructs in the algorithm. Places 2 and 4 are associated with the selection (*if... then ... else*) statements. Places 1 and 3, where we have merging control flows can be associated with labels where control can pass either in the normal sequential code interpretation or via *goto* statements. The interpretation of pairs of places (5, 6) and (11,12) has already been explained above. They are associated with the variables of the binary type, whose values are initialised according to the initial marking of the net.

What remains to be done is the algorithmic interpretation of the λ transitions. Those can be interpreted in terms of the value transition operators in the form: <previous value> -> <new value>. Namely, $\lambda0$: w:1->0, $\lambda1$: w:0->0, $\lambda2$: w:0->1, $\lambda3$: w:1->1. The "no change" actions for variable w ($\lambda1$ and $\lambda3$) are denoted with dashed arcs in Figure 10.

With these interpretations, we obtain the following algorithmic code for the write part process of Signal:

initialise *w:=1;*

wr1: write slot (w);

if *r=0* ***then*** *w:1->1;* ***goto*** *wr1*

else *(if r=1) w:1->0;* ***goto*** *wr0;*

wr0: write slot (w);

if *r=1* ***then*** *w:0->0;* ***goto*** *wr0*

else *(if r=0) w:0->1;* ***goto*** *wr1;*

This code can be "optimised" to the one which uses the "assignment to constant" statements instead of the state transition ones (we have to use some "don't care" facts for this optimisation because these assignments allow the previous values to be either 0 or 1 in general):

initialise *w:=1;*

wr1: write slot (w);

// don't care fact: previous value of w is not equal to 0 in this part

if *r=0* ***then*** *w:=1;* ***goto*** *wr1*

else *(if r=1) w:=0;* ***goto*** *wr0;*

wr0: write slot (w);

// don't care fact: previous value of w is not equal to 1 in this part

if *r=1* ***then*** *w:=0;* ***goto*** *wr0*

else *(if r=0) w:=1;* ***goto*** *wr1;*

This code can be further "optimized" to the one which uses the "variable assignment" statements instead of the assignments to a constant (note also the elimination of conditions and some of the *goto*'s because both guard cases, r=0 and r=1, are covered by the *if... then ... else* statements):

initialise *w:=1;*

wr: write slot (w);

w:=not(r); ***goto*** *wr;*

The final *goto* can of course be eliminated and a more elegant *do forever* can be used instead after the initialisation statement.

Performing similar mapping and optimising transformations on the read part, we arrive to the final script for the two-slot Signal algorithm:

Process Write:

initialise *w:=1;*

do forever

write slot (w);

w:=not(r);

Process Read:

initialise *r:=1;*

do forever

r:=not(r);

if *(w<>r)* ***then*** *read slot (r)* ***else*** *wait;*

6 Fewer or more slots

It is possible to implement a Signal ACM with only one slot, if the asynchrony requirement is relaxed to the point that data accesses by either side inhibits the other side. Such an implementation will, however, have a higher degree of synchronization between the two sides than prescribed by the basic data state specification of the Signal. In fact, for all types of ACMs in Table 1, if the asynchrony requirement is relaxed so that data accesses by both sides are mutually exclusive in time, implementations with only one slot can be found.

However, such one-slot implementations, while straightforward and almost trivial in the case of Channel and Pool ACMs, require more work for Signal and Message types because of the inherent asymmetry in them. Work is being carried out by the authors to develop single slot implementations of both Signal and Message in hardware.

Two slots are the minimum required to implement asynchronous Signal ACMs, but adding more slots may allow more asynchrony between the two sides. Here we further demonstrate the validity and usefulness of the method developed so far by a second case study of a Signal implemented with three data slots.

With more than two data slots, it may be possible to increase the asynchrony between the reader and the writer. The most obvious area where asynchrony can be further increased is additionally allowing the reader to start data accesses when the writer is in the process of data access. This can be captured by the Petri net fragment shown in Figure 11.

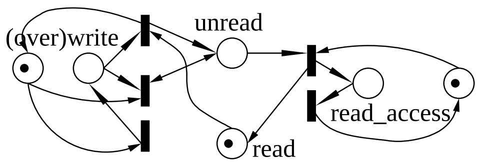

Figure 11: Fully non-blocking Signal.

Such bi-directional non-blocking is not possible with two slots. One counter example is given as follows: If reader occupies slot 0 with a single read access for long enough for the writer to go through more than one cycle, the writer will need to access slot 1 repeatedly. If when the reader comes back for another access the writer happens to be occupying slot 1 for another overwrite access, the reader must wait (slot 0 now containing previously read data and slot 1 containing half written data) although the writer has deposited (and then erased) newer data in slot 1. This means that the act of overwriting invalidates previously written data at the beginning of access rather than at the end of access. So there must be a third, indeterministic, data state during overwriting accesses.

In contrast, Figure 11 allows only two data states, 0 and 1, and stipulates that reader access must be able to start during an overwrite access. As the item of data being deposited by an overwrite data access is not valid for reading yet, a read access starting during this period must be directed to the data item generated by the previous write or overwrite access, which must be in a different slot. So for the specification in Figure 11 to be realizable, there must be a minimum of three data slots. In the worst case, one slot can be accessed by an overwrite access, another contains the item deposited by the last completed (over)write access, and the last one was accessed by the previous read access. Whether with three or more slots this specification can be realized remains to be seen. Here we attempt to synthesize an algorithmic solution with three data slots which conforms with Figure 11.

All arguments about data properties listed above for deriving the two-slot Signal implementation also apply here. The data freshness part of the specification needs to be modified. This is shown in Figure 12 which is a modification of item 3 in Figure 5.

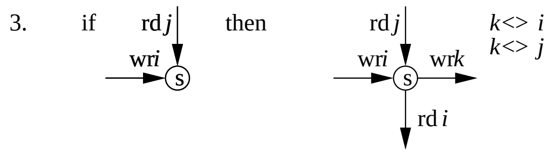

Figure 12: Data freshness constraint to state graph for three-slot Signal.

Here the desire of passing on the most up to date item of data is expressed by the full use of the three available slots. The writer is directed to the slot different from both the slot last occupied by the reader and that accessed by the previous write access. Therefore if the reader occupies one of the slots with one read access for a comparatively long time, the writer will alternate its accesses between the two remaining slots. This guarantees that when the reader comes back again for another access it will be directed to the slot containing the newest item of data deposited by the last completed write access. The reader does not wait if this slot is different from the one accessed by its previous cycle.

Based on this, a basic state graph for a three-slot Signal in the style of Figure 6 can be derived. This is shown in Figure 13. The dotted lines outline the mode of operation where the reader occupies one of the slots and the writer alternates between the other two. Otherwise all discussions given above for Figure 6 also apply here.

From this simple state graph, which is again not

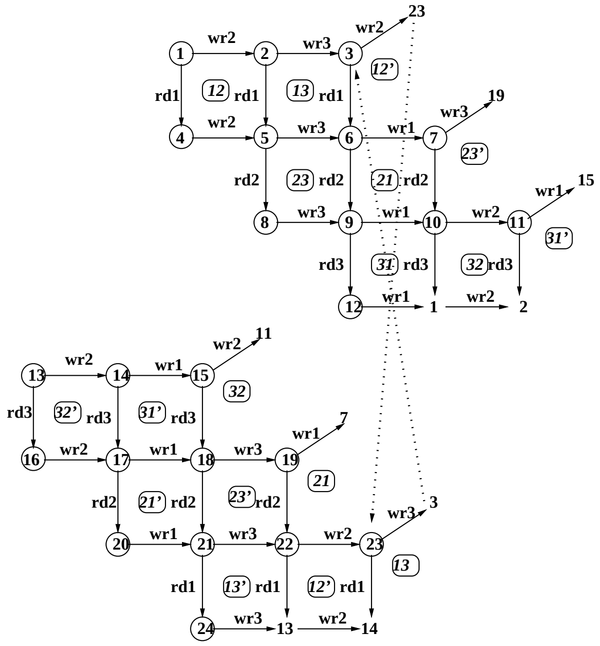

Figure 13: Simple state graph specification for three-slot Signal.

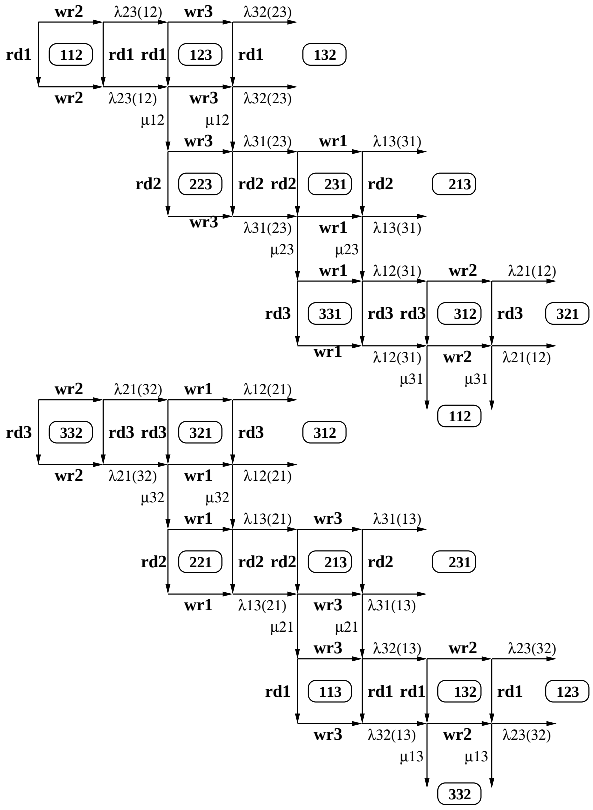

Figure 14: Refined state graph including silent actions.

suitable for direct use in synthesis, a "refined" version maintaining persistency for the wr and rd events has been developed. This, with the necessary silent actions added, is shown in Figure 14.

Techniques used in finding the two-slot Signal algorithm implementation have been employed to synthesize a three-slot Signal algorithm from the specification in Figure 14. This algorithm is given below:

Process Write:

initialise *w:=1; l:=3;*

do forever

write slot (w);

l:=w;

w:=¬(r, l);

Process Read:

initialise *r:=2;*

do forever

if *(l<>r)* ***then*** *r:=l* ***else*** *wait;*

read slot (r);

The additional control variable l (last) points to the slot accessed by the writer during its last completed data access. All three control variables are now ternary and the slots are labelled slot (1), slot (2) and slot (3). Additional text is provided in Figure 14 to facilitate understanding and enhance readability. Groups of four adjacent states in which the control variable values do not change are denoted by an oval shape around a three digit number such as 123, which means that for the four states surrounding this oval, $r = 1$, $l = 2$, and $w = 3$. This makes it much easier to trace the μ's and λ's. For instance, it can be quite clearly seen that $\mu 32$ is r:3->2.

7 Conclusions

A new technique for synthesis of communication mechanisms for heterogeneously timed systems has been described. The method has been used to (still manually though!) construct algorithms for an important mechanism called Signal, in its two and three slot variations. It would now be quite trivial for us to re-build a three slot Pool, from its formal state graph capture, the one which was taken as a given protocol for our hardware implementation reported in [14]. Even though we do not have the automatic procedure for synthesis of ACMs, this preliminary study has improved our knowledge about ACMs, their behaviour and taxonomy.

8 Acknowledgements

This work is supported by the EPSRC (Comfort project at Newcastle, GR/L93775). Our thanks go to our colleagues I.G, Clark, D.A. Fraser, A.C. Davies, D. Shang, A. Bystrov, N. Henderson, D.J. Kinniment, and A. Koelmans and to H. Simpson and E. Campbell of BAe for helpful discussions about ACMs, and to J. Cortadella, L. Lavagno, Ph. Darondeau, M. Kishinevsky, W. Vogler and M. Bednarczyk for discussions about Petri net theory of regions and insertion of silent actions.

References

[1] Lamport, L., “On interprocess communication parts I and II”, Distributed Computing, pp.77-101, vol.1, 1986.

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

[3] Kirosis, L.M., “Atomic multireader register”, Proc. 2nd Int. Workshop on Distributed Computing, Amsterdam, LNCS-312, pp.278-296, Springer Verlag, 1987.

[4] Simpson, H.R., Methodological and notational conventions in DORIS real time networks, Dynamics Division, BAe, February 1994.

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

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

[7] Simpson, H.R., “Correctness analysis of a class of asynchronous communication mechanisms”, IEE Proc. Vol. 139, Pt.E, No.1, PP.35-49, January 1992.

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

[9] Passerone, R., Rowson, J.A. and Sangiovanni-Vincentelli, A., “Automatic synthesis of interfaces between incompatible protocols”, Proc. DAC98, pp.8-13, Sanfrancisco, USA, 1998.

[10] Emerson, E.A. and Clarke, E.M., “Using branching time Temporal Logic to synthesize synchronization skeletons”, Science of Computer Programming, vol.2 , pp.241-266, North-Holland, 1982.

[11] Manna, Z. and Wolper P., “Synthesis of communicating processes from Temporal Logic specifications”, ACM Transactions on Programming Languages and Systems, Vol.6, No.1, pp.68-93, January 1984.

[12] Yakovlev, A., Xia, F. and Shang. D., “Synthesis and implementation of a signal-type asynchronous data communication mechanism”, Proc. Async2001, Salt Lake City, USA, March 2001.

[13] Badouel, E. and Darondeau, Ph., “Theory of regions”, Lectures in Petri nets I: Basic models, Advances in petri nets, LNCS, vol. 1491 (1998) pp. 529-586.

[14] Xia, F., Yakovlev, A., Shang, D., Bystrov, A., Koelmans, A. and Kinniment, D.J., “Asynchronous communication mechanisms using self-timed circuits”, Proc. Async2000, Eilat, Israel, pp. 150-159, April 2000.

[15] Nielsen, M., Rozenberg, G. and P.S. Thiagarajan, “Elementary transition systems”, Theoretical Computer Science, Vol. 96, pp. 3-33, 1992.

[16] Cortadella, J., Kishinevsky, M., Lavagno, L. and A. Yakovlev, “Deriving Petri nets from finite transition systems”, IEEE Trans. on Computers, Vol. 47, No. 8, pp. 859-882, Aug. 1998.

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

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

[19] Sgroi, M., Lavagno, L. and Sangiovanni-Vincentelli, A., “Formal model for embedded systems design”, IEEE Design and Test, Vol.17(2), pp.14-27, April-June 2000.

[20] Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L. and A. Yakovlev, “Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers”, IEICE Trans. Inf. and Syst., Vol. E80-D, No.3, pp. 315-325, March 1997.