Synthesis and implementation of a signal-type asynchronous data communication mechanism

A. Yakovlev, F. Xia and D. Shang
Department of Computing Science
University of Newcastle
Newcastle upon Tyne, NE1 7RU, UK
{Alex.Yakovlev,Fei.Xia,Delong.Shang}@ncl.ac.uk

Abstract

This paper describes the synthesis and hardware implementation of a signal-type asynchronous data communication mechanism (ACM). Such an ACM can be used in systems where a data-driven ("lazy") logic must be interfaced with a time-driven ("busy") environment. A new classification system for ACMs is introduced. The conceptual definition of the signal ACM (called simply "Signal") is refined using Petri net techniques. Based on this, a more precise, state graph specification of a two-slot Signal is then constructed. Using theory of regions, a Petri net specification of the ACM is synthesized from the state graph. The Petri net model is then translated into a hardware implementation, which is entered into Cadence tools. Simulation results show that the hardware does conform to the definitions and specifications. The techniques employed in this work are potentially useful in the development of an automated process of synthesising similar systems.

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 an intermediate data repository, usually in the form of shared memory, as shown schematically in Figure 1.

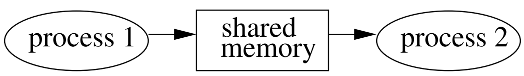

Figure 1. Passing data via shared memory.

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.

Many ACM protocols and systems for classifying these protocols have been proposed in the literature. One simple and elegant classification system for ACMs, developed by Simpson in [1] and [2], was based on the number of items of data in the ACM and its modification by the reader and writer accesses to the ACM. In this classification, the basic data state is the item of data in the ACM, which the writer and reader accesses modify by a system of "destructive" and "non-destructive" reading and writing. The scheme of this classification is shown in Table 1 which is obtained from [1]. In Table 1, "N-DR" stands for non-destructive reading, etc., and "Signal", "Pool" etc. are the names given by Simpson for the protocols that demonstrate the corresponding writing and reading rules.

Table 1. ACM protocols classification based on the destructive-ness of data accesses

	DR	N-DR
DW	Signal	Pool
N-DW	Channel	Constant

One of the most important real-time properties for ACMs is the amount of blocking/waiting the data state of the ACM demands of either accessing process. The data state cannot hold up the writer in a destructive writing scheme and the reader in a non-destructive reading scheme. The writer must wait when the ACM is full if writing is non-destructive and the reader must wait when the ACM is empty if reading is destructive.

Classifying ACMs based on whether data accesses destroy data in the ACM has limitations, however. One of the most significant shortcomings of this system lies with the role played by the type "Constant" which in effect does not allow any writing. This hardly qualifies as an ACM in the form of Figure 1. A modified classification system for ACMs is proposed in this paper to provide a more meaningful dual for the Signal ACM type.

Although the protocols developed by Simpson in [1] implied software implementations and most of the detailed ACM designs seen in the literature [3]-[6] indeed assume implementations in software, the significance of hardware designs and implementations have been gaining recognition more recently [7], [10], particularly in the area of building systems on chips with heterogeneous timing. These published ACMs are all of the Pool type, which can be used to implement truly atomic data transfer with full asynchrony for both the reader and writer. This makes the Pool very much suitable for transmitting reference data, as a kind of replacement for an analogue wire holding a variable value.

The dual to the Pool, the "Channel" ACM type, is the common no-loss buffer, most often with a FIFO arrangement, used widely in data communications.

A Pool is not suitable if, for instance, the reader process needs to continue only when there is new data available. This may be significant if the reader process is implemented with an asynchronous device where power savings can be realised by allowing it to wait whenever it can, or if the data being transmitted are of the interrupt and exception type.

For such applications, the "Signal" type ACM is more suitable. The basic Signal protocol does not hold up the writer but does the reader if the ACM is empty. The Pool and Signal are compared in Figure 2.

Apart from the new classification scheme, the major contributions of this paper are following:

(i) The construction of a Petri net specification of the two-slot Signal-type ACM using the synthesis methodology based on theory of regions in state graphs [14, 13]. The promotion of this methodology into the practice of designing protocols and asynchronous circuits is crucial in paving the way for the new generation of CAD tools capable of synthesising concurrent systems in general.

(ii) The implementation of the Petri net model as an asynchronous control circuit using a combination of techniques, such as direct translation of Petri nets into circuits based on David cells (this ensures transparency between behavioural specification and structural implementation) [12] and the use of relative timing (applied in a local way in order not to jeopardise the robustness of the overall selt-timed circuit) in order to improve speed of the control logic [19].

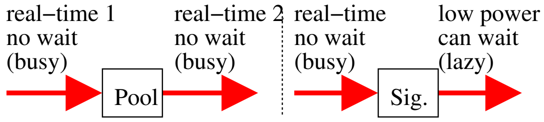

Figure 2. Comparing the applicabilities of the Pool and Signal.

In the following sections, the definition of the simplest Signal-type ACM, that with capacity 1, is refined by adding specifications on the effect of data accessing of one side to that of the other side, providing a better picture of the blocking characteristics. From this refined definition, a process of systematic synthesis is carried out by employing various state graph and Petri net [8] related techniques.

A hardware implementation of the resulting ACM is then carried out and extensive simulations verify the salient properties of the circuit.

2. New classification for ACMs

An ACM has a capacity, a non-negative integer constant, which is the number of data items it contains. Each data item an ACM contains is either read or unread, at any time. The basic data state of an ACM consists of the number of unread data items it contains.

Write data accesses are divided into writing and overwriting. Read data accesses are divided into reading and re-reading. Writing increases the data state by 1 (one more unread item in the ACM) and reading decreases it by 1 (one less unread item in the ACM) while overwriting and re-reading do not modify the data state. Overwriting may occur, if permitted by the ACM protocol, only when the ACM's data state is equal to its capacity, i.e. when all items of data in it are unread. Re-reading may occur, if permitted by the ACM protocol, only when the ACM's data state is 0, i.e. when none of the items of data in it is unread.

ACMs are classified according to whether overwriting and re-reading are permitted as shown in Table 2.

Table 2. ACM protocols classification based on overwriting and re-reading permission

	NRR	RR
NOW	Channel	Message
OW	Signal	Pool

In Table 2, NOW stands for no overwriting and NRR stands for no re-reading, etc. The Channel, Pool and Signal protocol names are inherited from the classification in Table 1 and are unchanged in effective specification. A new ACM type, "Message", is the dual of Signal.

In terms of the blocking of data accessing by the data state, if re-reading is permitted (RR) there is no holding up of the reader and if overwriting (OW) is permitted there is no holding up of the writer. If re-reading is not permitted (NRR), reader must wait when the data state is 0. If overwriting is not permitted (NOW), writer must wait when the data state equals the ACM's capacity.

Compared to Constant in Table 1, the new Message type is more general. In fact, Constant is a special case of Message where writing is disallowed entirely and reading is re-reading all the time.

This classification is deliberately non-specific for data item arrangements within ACMs with non-1 capacities in order to be as general as possible. Overwriting and re-reading in ACMs with capacities greater than 1 are treated as implementation issues and not explored in detail here. However, for ACMs with capacity 1, which is the main focus of this paper, the specific meanings of overwriting and re-reading are clear.

3. Definition of the 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 [11]) are also special cases.

This basic Signal definition is captured by the Petri net model in Figure 3. Here the places "0" and "1" are complementary to each other, one and only one being marked at all times.

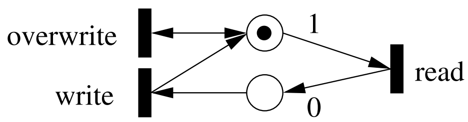

Figure 3. Basic definition of the Signal protocol.

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. E.g., the reader may or may not be required to wait while the writer is in the middle of an access because of the implementation. The model in Figure 3 is not specific about such distinctions.

In order to reflect the concept of non-atomicity of data accesses by the writer and reader, such accesses must be represented as distinctive states in the model. This is achieved by using the techniques introduced in [9]. Such a refinement is shown in Figure 4. 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.

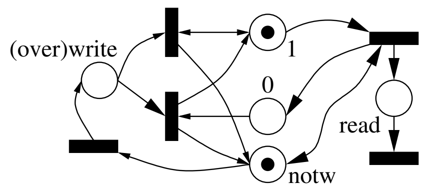

Figure 4. Signal with non-atomic writing.

By having only one token in the models, it is implied that the writer and reader are dealing with complete items of data one at a time and the integrity of these items of data is maintained, i.e. the reader is not supposed to obtain an item of data that is assembled from parts of different items provided by the writer, or otherwise corrupted.

Formally, the definition in Figure 4 specifies the following properties:

1. Data states and their updating:

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

At the start of a read data access, the Signal's data state is set to 0 (empty).

At the end of a write data access, the Signal's data state is set to 1, the item of data provided by this write data access being unread.

2. Conditional asynchrony for the reader:

A read data access may start only when the data state of the Signal is 1 and no write data access is occurring. 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 [3]:

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 [3]:

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 [3], 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 4 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 [7], [10]. 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.

4. State graph specification of Signal

The conceptual definition of Signal in Figure 4 cannot be used as a formal specification of the Signal protocol because it does not show the specifics of this protocol, such as how, for example, blocking on writing is avoided by using multiple 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. Let us first formulate those properties using the idea of states and transitions labelled by write and read actions. The reason for using state graph for specifying the ACM protocol instead of trying to construct the Petri net model directly is explained as follows. State graph modelling is much clearer for reasoning about the global properties of the system than that in Petri nets because a state graph is based on the concept of global states and interleaving 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 systems consist of a number of processes (write and read) and components (slots).

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 that lead to 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:

wr$i \notin P_s \Rightarrow$rd$j \notin N_s$, $\forall s$, i and j; 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 $= 0$).

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

2. Data coherence:

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

3. Data freshness:

wr$i \in P_s$&rd$j \in P_s \Rightarrow$ wr$j \in N_s$&rd$i \in N_s$, $\forall s$ and $i \neq j$; 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.

Condition 4 reflects the desire to avoid "retry loops" [6], so as to keep the solution simple for this first attempt. 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 4 and is shown in Figure 5. The initial state s1 is labelled with a big arrow. In this state, the ACM starts as "empty" with only writing to slot 1 enabled.

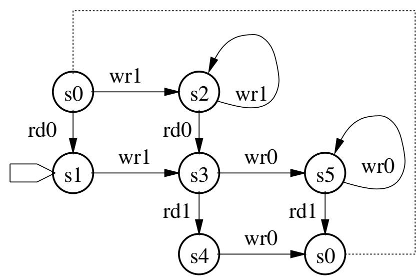

Figure 5. Simple state graph specification for a two-slot Signal (s0 is initial state).

In Figure 5, the dotted edge denotes that the two states at its ends are essentially the same state and the state graph is in closed form for readability.

All conditions mentioned above have been incorporated into the specification. For instance, at state s2 in Figure 5, rd1 cannot be in the next set because rd0 was in the next set at state s0 and has not featured in the next 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 [3]-[7]. The result of this refinement is shown in Figure 6. This refinement will also be needed in order to satisfy the separation conditions for Petri net synthesis, described in the following section.

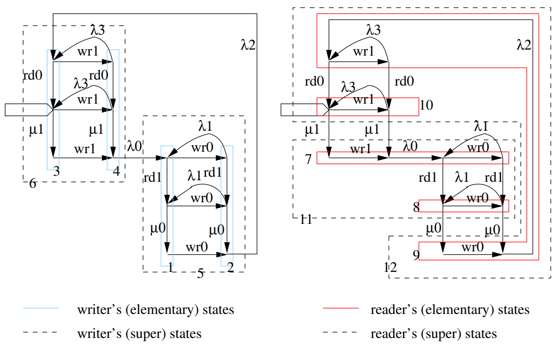

Figure 6. "Distributing" states between reader and writer parts using regions.

Both halves of Figure 6 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 (cf. next section). In Figure 6, the λ's denote silent actions performed in the write part 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 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).

5. Petri net specification synthesis

If we try to synthesise a hazard-free circuit in order to implement the state graph model in Figure 6, we need to encode this state graph and perform logic synthesis of the circuit. Unfortunately, this is not a trivial task not only because of the problem with race-free encoding of the symbolic states with binary signals, but also because, as one can easily observe from this graph, it involves non-trivial arbitration conditions in the states where $\lambda 1$ and $\lambda 3$ are enabled (viz., the action of the write part depends on the completion of the appropriate μ action of the read part). Our approach will follow a two-step procedure, described in [12], which firstly constructs a "distributed and concurrent system" (in the form of a Petri net) from the "sequential and centralised" description (in the state graph form), and then syntactically translates this Petri net into a self-timed control circuit. The first step is described in this section, the circuit synthesis in the following section.

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 [13]). 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 [14], 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/internal relationship. Depending on this 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 6 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 [13] 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 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 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 [13], 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 simply guarantee the state and event separation properties and use only minimal regions (regions which are not unions of other regions) is implemented in the Petrify tool [17]. The result of such synthesis is shown in Figure 7.

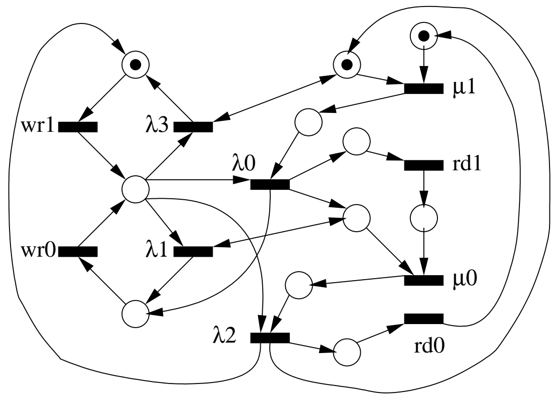

Figure 7. Petri net specification from Petrify.

This Petri net uses places that correspond to minimal regions in the state graph of Figure 6 (note that the regions depicted in Figure 6 by the dashed boxes are not all minimal - cf. regions 5 and 6). This net reflects the arbitration cases between $\lambda 3$ and $\lambda 0$, based on the state of the $\mu 1$ operation, and similarly between $\lambda 1$ and $\lambda 2$, based on the state of $\mu 0$. The techniques available for direct translation of Petri nets into self-timed circuits, described in [12], cannot be easily applied to this net, however, because of the problem of finding an adequate combination of hardware components to implement these arbitrations. We have looked at an alternative Petri net description that could be produced using the regions shown in Figure 6. The net generated by these regions is shown in Figure 8. This net also satisfies the synthesis problem requirements (its reachability graph is isomorphic to the state graph specification), and although it is not minimal in terms of the number of places used, it has a very interesting property from the point of view of its possible implementation, particularly in hardware. This net is essentially formed by two subsets of places, one belonging to the write actions (places 1 to 6) and the other to read actions (places 7 to 12). These two (finite-state-machine) subnets interact by using bidirectional arcs, which are also known as test or read arcs.

This way we have effectively decomposed the system into two parts 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 7 and 8 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 steering 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. This Petri net can be used to derive the circuit implementation, following the techniques of [12].

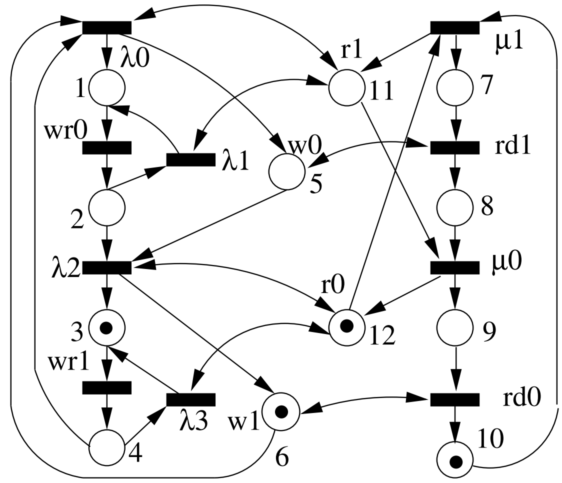

Figure 8. Petri net specification of a two-slot Signal.

6. Circuit synthesis

In these two sections we outline the process of converting the Petri net model of the Signal into an asynchronous circuit. The fact that the Petri net already captures the notion of a "decomposed state" in its places is exteremly important because we can exploit this distribution in the "net-to-circuit" translation.

Our first "sketch" of the Signal's circuit implementation, which is basically a homomorphic translation of the Petri net in Figure 8, is shown in Figure 9. The circuit is built with two-phase (event-based) signalling in mind (see [15]). It consists of the control skeleton part and the operational part, involving the write and read operations and the latches that implement variables w and r. These latches are equipped with pairs of control signals for set and reset handshakes, labelled "+" and "–" (note that these handshakes ensure that both setting and resetting of w and r are properly acknowledged). The latches also produce "dual-rail" signals, r0 (when r=0) and r1 (r=1) for r, and similarly for w. The write control part (see labelling of wires by numbers 1-4) is obtained by simply associating places 1 and 3, whose input transitions fire in a mutually exclusive manner, with XOR gates, and places 2 and 4 with requests to two 'arb' blocks, which are arbiters for sampling the (potentially changing) levels on signals r0 and r1, and depending on the 'r0=true' or 'r1=true' state of the test, generate one of the event-based outputs corresponding to the appropriate λ transition. These signals then either activate the appropriate write operation, either wr0 and wr1, depending on which of the two slots is supposed to be written, or send a request to toggle (set or reset) the state of w. In the read part, the control flow is very simple (note wire labels 7-11 to indicate correspondence with appropriate places in the Petri net) and it does not need arbitration. It awaits, at one of the event-based TL (cf. transparent latch) gates (initially, it is the left one, associated with the initial position of the token in the read part at r=1), the arrival of the condition w0 (when w=0) to be at 1 and then activates the rd1 operation, followed by the resetting of the r variable. After that it performs similar acton with r0 as soon as the w signal becomes 1.

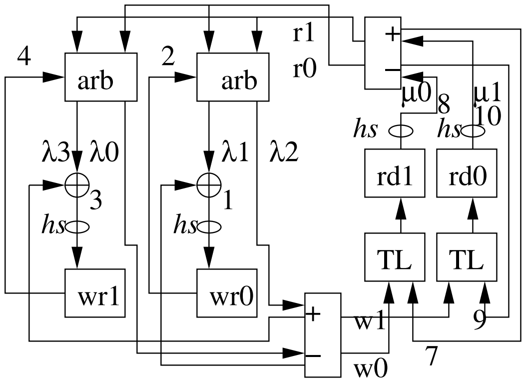

Figure 9. Block diagram of first circuit design.

This implementation is very schematic. In order to build the complete circuit for the Signal from it, it needs to be refined by providing interconnections with the two environment handshakes, write's request and ack and read's request and ack, which are implicit in this circuit. Those two handshakes could be created by breaking the wires that are indicated in Figure 8 by the ovals labelled with *hs*. Of course, to form just a single req/ack pair on each side, those double pairs need to be suitably multiplexed by using known two-phase elements, such as e.g. CALL from [15].

We have also studied, at greater length (believing that this will give us a faster implementation), another circuit translation of the Petri net model, the one based on four-phase signalling. This translation method, described in [12], is based on the idea of a 'one-hot-encoded' implementation of the Petri net model, in which places of the Petri net are associated with memory elements (called David cells, cf. [16]) of the control circuit. This approach differs from the STG-based logic synthesis with its 'compact' state coding used in Petrify [17]. Our attempt to follow the latter has resulted in overly complex gates, whose simple gate decomposition requires to sacrifice speed-independence significantly, and thus cannot guarantee the desired level of robustness. Furthermore, we believe that this design example will provide us with a good benchmark with which we can demonstrate the potential for a new synthesis from Petri nets, leading to future improvments in Petrify, proposed in [18]. In the remainder of the paper, we will concentrate on this design of the Signal ACM, which is eventually brought to a CMOS implementation using Cadence.

7. Hardware implementation and simulation results

The structural idea of the circuit implementation for Signal is shown in Figure 10. The read part is conceptually simpler than the write part and its description is left out. The write part, whose Petri net specification can be traced back to Figure 8, is shown in Figure 11. It consists of a set of David cells (showed in bold) to store the distributed state of the control and blocks representing the controlled logic. The controlled (operational) logic is simply inserted between the cells, by breaking the wire that signals the next cell about the arrival of the token. Note, e.g., the insertion of blocks wr1 and wr0 after cells dc0 and dc1. Note also that the environment itself is 'inserted' between cells (as handshake "done-wr"). Let's consider first the David cell part.

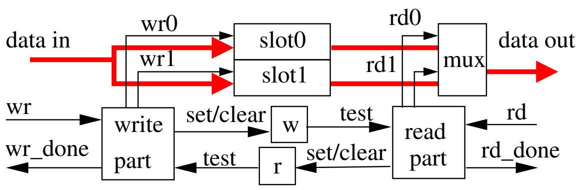

Figure 10. Block diagram of Signal circuit implementation.

These cells, built essentially around SR flip-flops (cf. Figure 13), represent the marking of the corresponding places in the Petri net (the absence of a token in place is associated with state 01 in the flip-flop, the presence with state 10). The labels of such places are shown in circles. In particular, blocks odc0 and 0dc1 stand for the cells corresponding to places 2 and 4. Block 2dc encapsulates two cells that model places 1 and 3 (the internal logic structure of this block is shown in Figure 12). The two pairs of sdc cells, labelled with 43 and 41, 21 and 23 respectively, are added in order to implement the appropriate branching of a token from place 4 either to place 3 or to 1 (similarly, for a token from place 2 to either place 1 or 3) depending on the arbitration decision made in the 'sync' block, which samples the value of the output r (rbar) of the binary variable 'read'. Such a sampling corresponds to testing the marking of places 11 and 12 (by read arcs) by transitions $\lambda 3$ and $\lambda 0$.

The cells named sdc are all built using a simple logic structure shown in Figure 13; they model the 'linear' pass of the control token. The cells named odc0 and odc1 are slightly more complex and allow merge of acknowledgements from two mutually exclusive branches of token flow (from places 4 and 2). For example, cell odc1 (for place 4) can either be reset to 0 after the pushing of the token back to place 3 (via the sdc cell 43) or to place 1 (via sdc cell 41), the latter also involving the execution of setw (event $\lambda 2$ in the net). This

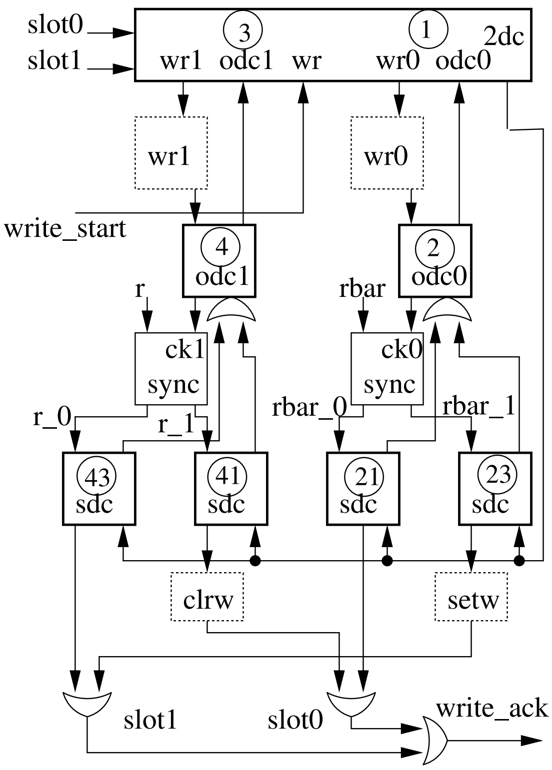

Figure 11. Circuit implementation of the write part using David cells.

is shown schematically by depicting images of OR gate at the resetting input of cell odc1, which collects ack signals from the sdc's 43 and 41. Implementing such a more complex reset function requires the use of 3-input NANDs in place of element xb Figure 13.

The operation of the control logic based on David cells can be visualised by performing a sequence of transitions on the cells signals as shown in Figure 13. The arrival of the token corresponds to the left-hand side handshake request w going to 0.

The dotted connection (from outa) in all David cells should be disregarded when the circuit is built maximally speed-independent. However adding such a connection would introduce a relatively 'mild' delay dependency (cf. relative timing [19]), concerned with the fact that the return of x back to its quiescent state (0) is left

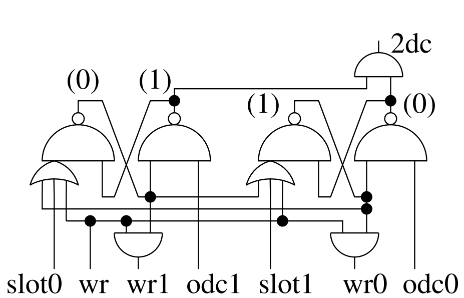

Figure 12. Logic for the 2dc block

unacknowledged. The gain from this is that transitions xb+, x- in the above sequence are executed concurrently with the forward propagation of the token. Under realistic delays in the gates, even with a zero delay for the controlled operations between the cells, there is ample time to complete those transitions (reset of the token) before the 'front' of the new token comes back through the control loop.

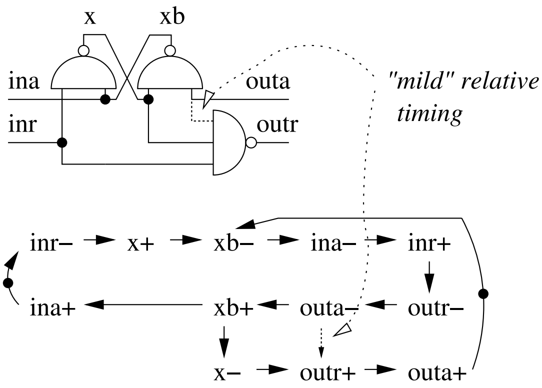

Figure 13. Simple David cell (with extra wire for a 'mild' relative timing assumption).

A bit more 'aggressive' relative timing is applied when the 'front' of the token is allowed to propagate forward as soon as it has been recorded in the cell, without even waiting for the completion of the preceding handshake. Again, under realistic delays, and assuming that the operational part takes at least a couple of inverter delays, this should be sufficiently robust. The modified logic of the case of a simple cell (linear transfer) is shown in Figure 14. Here the delay of passing the token through the cell is absolutely minimal - it takes two inversions, x and xb, from inr- to outr-.

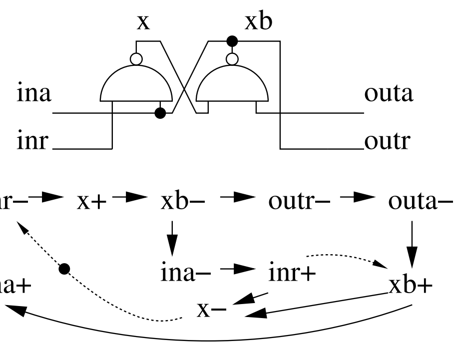

Figure 14. Simple David cell with 'aggressive' relative timing

Extensive analogue simulations (for 0.6μm CMOS technology) have been conducted on two circuit implementations, one maximally speed-independent (SI) using simple David cells (Figure 13) and the other with aggressive relative timing (RT) using the circuit in Figure 14 when possible. Both implementations have been confirmed qualitatively to satisfy the specifications. Their relevant full (write, read) cycle times are given in Table 3. These times are measured between the adjacent rising edges of the write requests (write_start+ $\rightarrow$ write_start+) and read requests (read_start+ $\rightarrow$ read_start+). For writes two modes considered are those that do (SR) and do not involve (NSR) switching (set or reset) the value of flag w. For reads, which can be naturally blocked by absence of new data, only one mode, called "no waiting" for new data (NW), is considered.

Table 3. Cycle times (in ns)

type	Write		Read
	NSR	SR	NW
SI	9.0	10.4	9.0
RT	4.8	6.3	6.6

In systems with true timing heterogeneity, metastability in the 'sync' arbiters is unavoidable if one or both sites of an ACM are permitted unlimited access. In our implementation, metastability is contained within the 'sync' arbiter blocks (Figure 15) and analogue simulations (Figure 16) confirm that it does not propagate through the system. Note that, after write request is set to high and ck0 is generated in order to sample the current value of variable r, the latter is making its transition from high to low (see the "input to mutex" window). This puts the SR flip-flip (we use a standard mutex implementation due to Seitz [20]) into a metastable state (see the upper window), which is eventually resolved in favour of rbar_0, i.e., the old value (high) of r. The outputs of the mutex, rbar_0 and rbar_1, produce clean edges. These signals are then used to generate ack signals leading to the control logic, which in this case passes control to one of the David cells sdc, labelled 21 and 23, to perform either nothing or 'setw' (setting w to 1), followed by the rise of the write_ack signal.

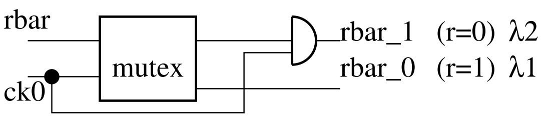

Figure 15. Implementation of a 4-phase 'sync'

8. Message, the dual of Signal

The new ACM classification of Table 2 made it possible to define a more useful dual for the Signal. This is the new Message type which permits re-reading but not overwriting.

The Message therefore does not hold up the reader, but does hold up the writer when the data state is equal

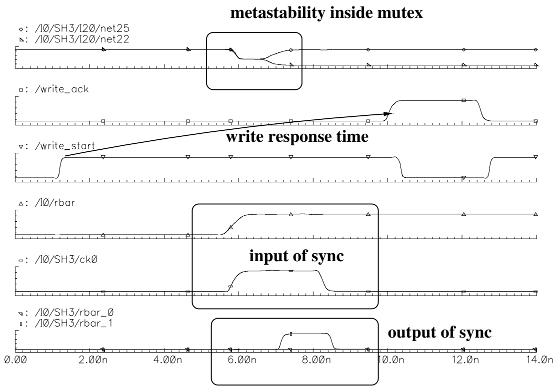

Figure 16. Illustrating metastability

to the ACM's capacity (all data items in the ACM unread). Such an ACM is useful when loss of data items is not permitted and when the reader process must be given temporal independence from the ACM's data state. Roughly speeking, from the writer side, new "Messages" may not be generated if previous ones have not been received, as compared with "Signals" being generated regardless of if previous ones have been received. The reader, on the other hand, won't wait for a new Message to appear but will re-use the previous one if needed, but will stop and wait for a new Signal if one is not available.

Obviously, the Message is a simple mirror image of the Signal. The definition models in Figures 3-4 can be simply reversed to form similar definitions for a 1-capacity Message, and it can be said that the implementation above is also that of a simple Message with full asynchronism for the reader. It just need to be connected to the writer and reader processes the other way round from the Signal.

9. Conclusion

A new system of classifying asynchronous data communication mechanisms has been proposed. Techniques for the synthesis and implementation of ACM protocols, only partially formalised and to a large extent unautomated, have been presented here. The overall design proceeded along the following steps: (1) conceptual definition (using Petri net 'sketches'); (2) construction of the basic protocol specification (using a state graph); (3) refining the state graph with silent actions; (4) synthesis of a Petri net specification (using regions); (5) translation of the Petri net model into a circuits using David cells; and (6) entering the design into Cadence and simulation. The model transformation steps (2-4) are currently only supported to a very limited extent by Petrify synthesis software. The refinement with silent events, to make the state graph synthesizable into a Petri net of a given class, is a very challenging theoretical problem and more research is needed here. The direct translation of Petri nets into David cells and subsequent optimisation with relative timing is another problem to be tackled in the future.

A simple Signal ACM, using only two data slots, has been synthesised and implemented in hardware. The same ACM, if connected in reverse, can also serve as a Message, the dual of Signal in the new classification. With the previous results reported in [7] and [10], examples of all the non-trivial ACMs in the new classification have been implemented. This further enriches the knowledge about ACMs and their hardware implementations. The latter provide an efficient and robust means of on-chip interfacing between data-driven, and thus power-saving, logic and a time-driven environment or between subsystems with independent temporal domains in highly timing heterogeneous systems.

10. Acknowledgement

This work is supported by the EPSRC (Comfort project at Newcastle, GR/L93775). Many thanks to Alex Bystrov and other members of the Newcastle's Async Systems Lab, to Ian Clark and other colleagues at King's College London for their useful feedback, to Philippe Darondeau for stimulating discussions about synthesis of concurrent systems, and last but not least to the anonymous referees for their helpful comments.

References

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

[2] 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.

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

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

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

[6] 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.

[7] 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.

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

[9] 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.

[10] Shang D., Xia, F. and Yakovlev, A., "An implementation of a three-slot asynchronous communication mechanism using self-timed circuits", Proc. AINT'2000, as in [2], pp.37-44.

[11] 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.

[12] Yakovlev, A. and A. Koelmans, "Petri nets and hardware design", Lectures on Petri Nets II: Applications. Advances in Petri nets, LNCS-1492, pp.154-236, Springer-Verlag, 1998 .

[13] 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.

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

[15] Sutherland, I.E., "Micropipelines", Commun. ACM., vol. 32, no.6, pp. 720-738, June 1989 (A Turing Award Lecture).

[16] David, R., "Modular design of asynchronous circuits defined by graphs", IEEE Trans on Comp., Vo. C-26, no.8, pp. 727-737, Aug. 1977.

[17] 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.

[18] Blunno, I., Bystrov, A., Carmona, J., Cortadella, J., Lavagno L., and A. Yakovlev, "Direct synthesis of large-scale asynchronous controllers using a Petri-net based approach", Handouts of Fourth ACiD-WG workshop, Grenoble, Jan. 2000 (http://time-cmp.imag.fr/tima/cis/cis.html).

[19] Stevens, K., Ginosar, R., and S. Rotem, "Relative Timing", Proc. IEEE Symp. on Adv. Res. in Async. Cir. and Syst. (ASYNC'99), Barcelona, Spain, April 1999, IEEE Comp. Soc., pp. 208-218.

[20] Seitz, C.L., "Ideas about arbiters", LAMBDA, First Quarter 1980, pp.10-14.