Data Communication Mechanisms for Systems with Heterogeneous Timing

Ian G. Clark* Fei Xia Alex V. Yakovlev Delong Shang

School of Electrical, Electronic and Computer Engineering,
Merz Court, Newcastle University,
Newcastle-upon-Tyne, NE1 7RU, England.
IGClark@iee.org, {Fei.Xia; Alex.Yakovlev; Delong.Shang}@ncl.ac.uk

Abstract

In this paper Asynchronous Communication Mechanisms (ACMs) are discussed in several guises based upon whether or not the heterogeneously timed processes which communicate through an ACM may be forced to wait or not. A classification method is presented based upon the temporal effects of this communication. The mechanisms' application to event driven and self motivated processing is discussed, revealing an interesting application as an interface between real-time and low power processing units.

1 Introduction

Many applications (e.g. portable equipment, control or signal processing systems, etc.) are characterized by the requirements to guarantee real-time regimes in some parts and power ecomony in others. From the system implementation point of view, the ITRS roadmap (http://public.itrs.net/) indicates that systems-on-chip (SoC) are increasingly becoming heterogeneous in their behaviour, including mixed analogue-discrete components, time-driven and power-saving subsystems. Another feature of new systems is that their design is becoming increasingly communication-centric. Consequently, the need for data interfaces between subsystems and processes with different temporal and power requirements, with emphasis on real-time and energy consumption characteristics, has become urgent.

Wilfred Pinfold of the Intel Microprocessor Research Lab says that by the end of this decade, when Intel expects to be producing billion transistor devices, today's essentially homogeneous microprocessor market will have to become much more diverse, characterised by multiple heterogeneous designs, each optimised for the requirements of different application segments [5]. This will mean that systems are more complex but with the added advantage that "Heterogeneous systems can use 'best-in-class' processors" (for cost, power, EMI, thermal budget), and can also support lagacy software [4].

Practical examples of systems with heterogeneous timing and power-saving requirements include embedded real-time control and signal processing systems, used in aerospace, automotive, telecom and other applications. Constructing such systems as digital networks with inherent heterogeneous timing conditions (called "hets" from "heterogeneously timed nets") in which computational elements interact through ACMs holds many attractions. These include a more formal recognition of system timing heterogenity, a network and potentially hierarchical view of the system composed of clearly active and passive elements, and direct representation of data flow through the system, all of which facilitates both top-down and bottom-up design approaches and the assembling of systems from sub-systems and elements. Although the idea of building hets does not itself stipulate that such systems are necessarily implemented in hardware, our perception is that this approach may be particularly effective in building the new generation of VLSI systems, or SoCs, where demands for higher speed, global asynchronisation and power savings come into one complex interplay. As the scope for the use of SoCs will continue to grow unabating, particularly as embedded systems with interfaces to real-time and analogue environments, serious re-thinking of fundamental principles of system design is needed to make the most of the future semiconductor technology. This will need a wide range of communication elements adequately geared to the types of data being exchanged in the system, and to the dynamic properties of the data, such as uniqueness, continuity, differentiatability etc.

Power considerations can also mean that subsystems in distributed systems cannot all be in the same timing domain. For instance, subsystems may have bounded energy resources, and it may be desirable for them to enter a sleeping mode if there is no change (or no need for change) in its input/output data. It is also possible to have subsystems where the operational speed may be a function of local energy supply conditions (a subsystem can be slowed down when its energy reserve becomes low) and not dependent on other subsystems with whom they are communicating.

Clearly, such interfaces need to adjust to the inherent contradiction between the different desired temporal and power characteristics of its client processes. Therefore, radical changes need to be made at the level of data communication mechanisms, such as the use of asynchronous/clock-free circuits and more direct use of hardware (for temporal predictability and power saving) instead of layers of software to support communication protocols.

In addition, in recent years, "soft-computing" technologies such as fuzzy logic and neural networks have become increasingly popular, particularly in the context of embedded systems. The unifying characteristic of these technologies is their somewhat relaxed view of the impor-

*Also with the School of Computing and Information Systems, Kingston University, Kingston-upon-Thames, Surrey, KT1 2EE, England.

tance of the precision of individual items of data. The increased robustness of such techniques makes it possible to soften the requirement on data precision, leading to various advantages in implementation.

Similar levels of attention, however, has not been paid to the possible "softening" of the requirement on temporal relations between processing elements in complex systems. Even for soft-computing technologies, it has almost invariably been assumed that systems operate under global synchrony. In addition, in established soft-computing technologies, the softening of data happens at processing units, while data communications between processing units are assumed to be fully conventionally precise. In other words, communication units are still required to treat the data being passed through them as inviolable, although from the system-wide point of view the precision of individual data items is not important. This does not present a problem if the system is assumed to operate under global synchrony.

Requiring that every item of data generated by a processing unit reach its destination and the intended receiver of a data stream receive an exact copy of the stream generated for it without any error makes it impossible not to have some synchronization between the generating and receiving processes. This implies that all processing units in a system may be required to be temporally related to one another. This is considerably relaxed from global synchrony or a single clock, but is still too restrictive for many real-time systems.

If, on the other hand, data softening can be applied to the loss and repetition of items, processing units can be given temporal independence for the purpose of real-time safety, power savings or making SoC implementations more practical. In other words, softening synchrony via the allowance of data loss or repetition holds attractions for both real-time operations potentially catering to wait-free operations of components and very high scale system integration where unified clocks are impractical.

These application considerations also support the concept of hets. Figure 1 shows the hets concept of building systems with ACMs connecting active computational elements.

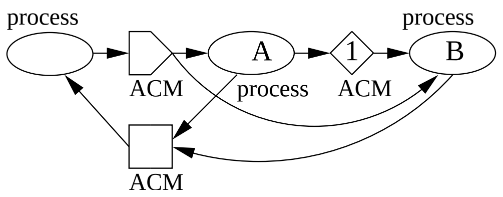

Figure 1: A het is a digital network with computational elements connected through ACMs.

This paper centers around the notion of ACMs which are the key to constructing hets. We present a classification of ACMs based on their temporal relationships with their access processes. We demonstrate the feasibility of building hets in hardware by presenting a systematic approach to ACMs, including methods of definition, specification, design and implementation at various levels, and analysis. We are especially interested in ACMs that are capable of interfacing an active element with self-motivated timing with another active element.

Previous work: The MASCOT (Modular Approach to Software Construction Operation and Test) design methodology, developed primarily at BAe and RSRE [17] for constructing real-time distributed systems of a safety-critical nature, provides initial stimulus for studying systems with multiple self-motivated timing domains (sometimes called "timing motive powers"), and forms a methodological basis for the investigation of hets. Up to now, the MASCOT method has been used for building systems which consist of off-the-shelf processor blocks running microkernels, and most of their communication mechanisms implemented in software, or if in hardware, then using sufficiently large "timing reserves" (use of the so called fundamental mode, which allows the circuit to stabilise before the next input stimuli are applied).

More recently, Metropolis, a design environment for heterogeneous embedded systems, has been reported [1]. The Metropolis method emphasizes the separation of computation and communication as orthogonal aspects of system design, and the decoupling of such orthogonal aspects over a set of abstraction levels is used to enhance the reusability of components. The explicit recognition of system hetergenity, including temporal hetergenity, and the emphasis on the orthoganality between computation and communication are similar to concerns in hets and ACMs, although detailed communication elements investigation has not been reported yet in Metropolis. Moses is another recently reported tool, which uses Petri nets and can model and simulate heterogeneous systems [12].

2 Asynchronous Communication Mechanism Taxonomy

In heterogeneously timed systems, data interfaces may 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 synchronized, it is often necessary to pass the data through some intermediate data repository, usually in the form of shared memory, as schematically shown in part of Figure 1. The writing process is labelled 'A', the shared memory is labelled '1' and the reading process is labelled 'B'.

An ACM is a scheme which manages the transfer of data between two or more processes not necessarily synchronized for the purpose of data transfer. Here we assume that there are only two processes involved. It is also assumed that the data being passed consists of a stream of individual items of a given type. It is further 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 access to the ACM. The provider of data is known as the "writer" and the user of data is known as the "reader" of the ACM.

The aim of the ACM is to pass data between the two processes without interference according to given data properties (see later) and according to certain temporal rules based upon the type of communication required. The different communication types are based upon whether the writer or reader is required to wait or not, see Figure 2. This waiting is dependant on the data state of the ACM. This is a modification to a classification system previously given by Simpson [20, 21], which was based upon destructive and non-destructive operations between the communicating processes and the ACM.

In Simpson's original ACM taxonomy if the writing of data was non-destructive and the reading of data was also non-destructive then the resulting communication protocol was called a ‘Constant’. Data could be written into a Constant once, normally during the system set up or start, and could never be modified. In the modified classification Constant is replaced by ‘Message’. The ‘Message’ protocol is more relevant for interfacing between real-time (self motivated) and low-power (event motivated) subsystems.

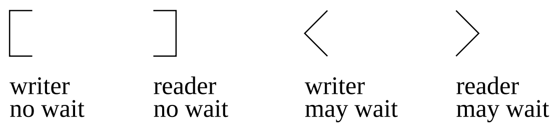

Figure 2: Symbols for temporal relations between reader/writer and ACM.

Intuitively, without going into the details of specification and implementation, ACMs can be classified into four groups based on the combination of may require waiting/may not require waiting for the two access processes.

There is an implied desire, for any ACM, that as much of the information from the writer should be passed on to the reader as possible, once the basic asynchrony specifications of the ACM are satisfied. This means that when a “no waiting” requirement implies imperfection in the passing of data, a method must be found to minimize such imperfection. There is also an implied desire, for any ACM, that as much asynchrony as possible is accorded to the writer and the reader within the specification. This means that any waiting, in a “may require waiting” specification, should be invoked only when absolutely necessary.

For the writer, the question of waiting or not waiting therefore comes up only when an ACM is full of items of data which have not been read. If it is important that every item of data from the writer must eventually reach the reader, the writer must be required to wait in this situation. On the other hand, if the writer process cannot be obliged to wait for data communication purposes, data must be discarded or lost in this situation. Straight discarding of the current item of data by the writer process is not a good solution, since this item of data, being of newer vintage than those already in the ACM, should normally be regarded as more important (cf. freshness property defined later in this section). It is, therefore, a better idea if overwriting should occur, in other words, some item already in the ACM should be replaced by the current item of data the writer has ready to write.

Similarly, when an ACM contains no previously unread data items, in order for its reader process not to be blocked by the data state, the reader must be allowed to go away empty handed or with some item of data it has acquired during a previous access. The best option here, in view of the desire of generating some data passage with this visit of the reader, is to allow the reader to re-read the item of data which it obtained during the immediately previous data access.

Overwriting and re-reading are also superior to straight discarding and doing nothing in the sense that they require essentially the same time to be carried out, in any realistic implementation, as normal writing and reading. This means that a temporal consistency exists for ACM accesses for the reader and the writer. Such temporal consistency is crucial, or at least desirable, in many hard real-time, safety critical systems.

Permitting overwriting (but not necessarily requiring overwriting all the time) implies that the writer cannot be held up (blocked) by the data state, while permitting re-reading (but not necessarily requiring re-reading all the time) implies that the reader cannot be held up (blocked) by the data state. A new ACM classification system based on whether these actions are permitted by any ACM protocol is proposed below. It has full correspondence to the no waiting required/waiting may be required paradigm.

2.1 ACM classification:

Formally, 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 Figure 3.

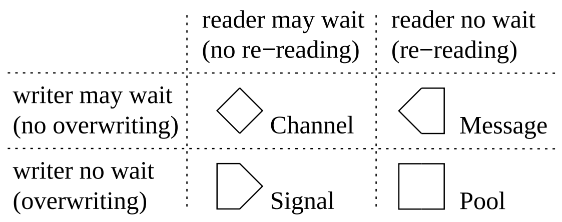

Figure 3: Graphic symbols for ACM types.

The Channel, Pool and Signal protocol names are inherited from Simpson’s classification in [21]. A new ACM type, “Message”, is introduced as the dual of Signal.

In terms of the data state blocking data access, if re-reading is permitted there is no holding up of the reader and if overwriting is permitted there is no holding up of the writer. If re-reading is not permitted, reader must wait when the data state is 0. If overwriting is not permitted, writer must wait when the data state equals the ACM’s capacity.

Traditional computer systems, if they do not interface to analogue or real-time environment, will only use Channels for communication, in the form of FIFO / LIFO / RAM buffers. This is because for these systems, asynchrony is secondary to the preservation of data. These solutions are without data loss and repetition so have to maintain some synchronisation. Typically, in such systems, data loss and repetition are treated as anomalies, therefore part of fault tolerance considerations. They are well studied in hardware implementation. The attention of this paper will be focused on Signal and Pool. Since Message is the dual of Signal, any work on Signal will also shed light on Message directly.

2.2 Practical significance of various types of ACMs:

In systems with heterogenous timing, there may exist active elements which are self-motivated in timing (i.e. the timing of such elements is entirely determined internally) and other elements which have reactive timing (i.e. the timing of such elements is dependent on their data communication interfaces). More complex situations such as a single element having a combination of self-motivated and reactive timing can also exist.

For any specific ACM, its reader and writer processes may belong to active elements which may need to have self-motivated or reactive timing with regard to this ACM. Self-motivated timing may be most useful when an active element is charged with real-time tasks, where precise timing and temporal predictability are important, or when an active element may need to have specific and controllable power and/or performance requirements and needs to be able to speed up or slow down without considering its relationship with an ACM. On the other hand, reactive timing may be useful when an active element is charged with tasks with enough temporal flexibility to accommodate waiting at an ACM to reduce data loss and/or repetition or to reduce power consumption (do nothing when no new data is available or needed).

In general, not requiring an access process to wait provides for the possibility of allowing that process to belong to a active element with self-motivated timing. On the other hand, (sometimes) requiring an access process to wait implies at least some degree of reactive timing exists for the active element to which the access process belongs. Therefore, Pool ACMs can be used to interface elements with self-motivated timing to each other, Signal and Message ACMs are best used to interface elements with self-motivated timing to elements with reactive timing, and Channel ACMs can only be used to interface elements with reactive timing to each other.

3 Data Properties

In addition to asynchrony, other important properties of ACMs exist and some of these have been investigated in previous work. The main data-passing properties include data coherence, data freshness, data sequence, data loss and data repetition. Other properties, such as power and hardware efficiency and temporal consistency are also important. Some of these have not been dealt with in detail.

Data coherence refers to the integrity of individual data items being passed through an ACM. Items of data obtained by the reader should not have been modified since their introduction by the writer. In other words, data items going through an ACM should retain their individual integrity or atomicity. If an item of data changes between the writer and the reader, data coherence is said to have been lost.

Data freshness refers to the desire for the reader to obtain the most up to date data item in an ACM that satisfies the particular protocol. For ACMs with capacity 1, this property is most easy to understand. For instance, with a capacity 1 Pool, any read access should obtain the data item introduced by the last write access to complete before it.

Data sequence refers to the desire that the reader should obtain data in the same order in which the writer produced it. It may appear that this property is redundant in light of data freshness above, however, when a writer is producing data far more quickly than a slow reader can acquire it, then there are instances where several pieces of data all seem to have valid freshness and data may be obtained out of order [3].

Data loss and *data repetition* are the results of overwriting and re-reading, respectively, and are inevitable consequences of no waiting and bounded ACM implementations. Data loss occurs when overwriting happens, causing one item of data which was introduced by the writer previously to be never available to the reader, in other words, lost in transit. Data repetition occurs when re-reading happens. In this case, the reader obtains an item of data it has already obtained in an earlier cycle. This type of analysis has been performed quantatively [10].

Power efficiency refers to the desire to minimise the power consumption of hardware implementations of ACMs. Hardware efficiency refers to the desire to implement any ACM with the smallest possible amount of silicon [3]. Depending on the implementation, these two properties may be directly related. Temporal consistency refers to the desire to have uniform expected temporal characteristics for a given writer or reader, i.e. all read accesses to an ACM from the same reader should require the same expected time to complete. Other well known properties such as latency and throughput may also be considered with regard to ACMs.

4 Metastability

Metastability [6, 11] is unavoidable in systems using latches between unsychronized processes (i.e. sampling data from an unknown temporal source). Some designers use metastability to their advantage [7]. Simpson calls this phenomenon 'dither' and claims that if enough time is allowed for the signals to settle then the output will either settle to its old value or its new one. This however requires that the designer leave delays for such settling (i.e. fundamental mode). In some systems this may be possible because, as in Simpson's MASCOT systems, the ACMs are controlled by software instructions which progress relatively slowly compared to raw hardware. Metastability can be modelled in descrete environments by having an 'M' state [2], and this can help to determine what effect a metastable state may have on a system.

If latches prone to metastability were used in the data path of the ACM the results could be dangerous. If one bit of a multi-bit word were to go metastable, the output could settle to a value which was neither the new value nor the old one (in Lamport's terminology this is known as a safe register [9]). If data coherence is to be maintained, metastability must not be allowed to happen in the data path. By employing multiple data slots[1] ACMs in the literature solve this problem. "Control variables" of the smallest granularity (binary or ternary) are used to steer the reader and writer so that they never simultaneously access the same slot. Asynchrony is thus removed from the data path to the control variables where the consequences of metastability are more easily contained.

Solutions published in the literature [22, 8, 18] use these techniques. These algorithms have been tested under sin-

[1] In the terminology of multi-slot ACMs [18], a "data slot" is a unique portion of the shared memory which may contain one item of data.

gle metastable events using Petri net modelling and analysis techniques [3]. If however shared latches in the control circuitry are protected from concurrent activity by an arbiter, allowing only one process access to a latch at any time, then a trade off between possible control metastability and possible response latency can be made. In some applications this may be unacceptable. An advantage of using such arbitration is the reduction in the number of slots required for successful implementation.

5 Modelling

In this section the various ACM types with capacity 1 are described with Petri net [13] models which are the basic state-transition definitions of these protocols. These top-level models define the various ACM protocols in sufficient detail so that what has been said informally earlier can be made unambiguous. They can also be helpful in system design and analysis at a level where the detailed implementations of ACMs are not important.

The basic definition of Signal treating read and write accesses as atomic is shown in Figure 4. In this definition, re-reading is not allowed, normal writing occurs when the data state is 1 and overwriting occurs when the data state is 0.

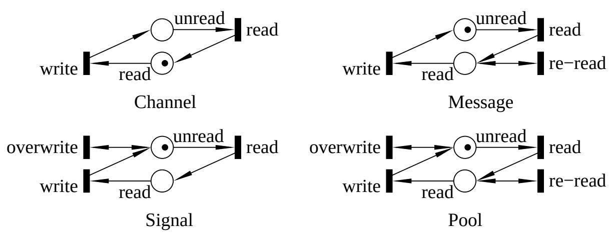

Figure 4: ACMs with atomic reading and writing.

From this definition it can be seen that Signal is most useful in connecting a writer process which exists in an active element with self-motivated timing to a reader process which exists in an active element with reactive timing. By disallowing re-reading, the reader side of the protocol puts the priority on the data passing properties. By allowing overwriting, the writer side of the protocol puts the priority on asynchrony. Thus the writer cannot be blocked by the data state while the reader can. In other words, the writer enjoys temporal decoupling from the state of the ACM. Data passing through a Signal may be of the interrupt/exception type, where the generator of these requests must be accorded temporal independence, and a new request would superceed any previous requests still not handled.

At the same basic level, treating read and write accesses as atomic, the Message protocol is also defined in Figure 4. This is a mirror image of the basic model of Signal, showing that Message is the dual of Signal. Message is therefore most useful in connecting a reader process which exists in an active element with self-motivated timing to a writer process which exists in an active element with reactive timing. A new "message" type data item would not be generated by the writer if previous ones have not been acted on. This is different from "signals" which are generated regardless of whether previous ones have been dealt with by the reader.

Similarly, Pool type ACMs may be specified between writer and reader processes both existing in elements with self-motivated timing or when it is otherwise a good idea to not allow data communications to affect the timing of either communicating side, while Channel type ACMs may be specified when the integrety of the data item stream being transmitted is paramount and the timing of both reading and writing processes can be made reactive to accomodate this. The Petri net definitions of both Channel and Pool can also be found in Figure 4.

6 Algorithms

We have designed Pool and Signal ACMs algorithms in a number of different ways. Here two examples are presented to illustrate our method of ACM analysis and verification.

6.1 A Pool ACM using three data slots

The models in Figure 4 are sufficient to describe the basic relations between reader and writer, but they treat the read and write data accesses as atomic actions. This is not true at the hardware level if a single data item is not of very small size. If these accesses are regarded as non-atomic processes that take non-zero time, a Pool should be modelled by the PN fragment shown in Figure 5. This specification gives total temporal independence to both reader and writer with regard to the ACM and each other.

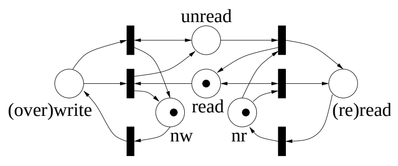

Figure 5: Non-atomic Pool model

A Pool specified by Figure 5 cannot be implemented with only one data slot and still provide data coherence, because simultaneous reading and writing accesses of multi-bit data items from the same data slot are likely to result in the modification of data after leaving the writer.

It has been shown [18, 23, 3] that a fully asynchronous Pool with capacity 1 can only be implemented using three or more data slots. Here we present a Pool ACM algorithm using three data slots found in [18] shown here in Figure 6.

In ACM implementations using multiple data slots, the number of data slots should not be confused with the capacity of ACMs. In general, the number of slots needs to be more than the capacity of a certain ACM if any asynchrony between reader and writer is desired. This does not necessarily present conflicts. For instance, in a capacity 1 Pool ACM correctly implemented with three slots, at any time, only one of these slots contains the current data item in the ACM and the others are used for avoiding the simultaneous reading and writing of the same physical memory. In order for an ACM to be implemented correctly with more slots than its capacity, the vital property of data freshness must be maintained.

In the algorithm in Figure 6, the data in passage is held in three data slots labelled slot 1 to slot 3. The control variables n, l, and r are ternary. It is implied in this algorithm that within the reader and writer processes,

```
writer                    reader

wr: write slot $n$        r0: $r := l$
w0: $l := n$              rd: read slot $r$
w1: $n := \neg(l, r)$
```

Figure 6: Algorithm of 3-slot Pool.

the statements must be executed in the order specified and any statement may not start without the previous one having completed.

The 'differ' statement w1: $n := \neg(l, r)$ means that variable n is assigned a value different from the current values of both l and r. In practice, this can be done by using the matrix $\neg(l,r)$ = ((2,3,2), (3,3,1), (2,1,1)). Such a "lookup table" can be easily implemented in hardware if desired.

Whether any ACM implementation actually conforms to the specification can be established using theoretical analysis employing Petri net models which highlight the important asynchrony and data properties such as the requirement of waiting, data coherence and data freshness. These kinds of modelling and analysis techniques have been described in previous work [3, 23, 25]. Here we will illustrate these techniques by modelling and analysing the ACM algorithms presented in this paper.

Algorithms like the 3-slot Pool in Figure 6 consist of two single-thread cyclic processes, which can be viewed as finite state machines (FSMs). We use the standard approach to derive Petri net models for such FSMs at the top level, resulting in the Petri net model of the 3-slot Pool algorithm shown in Figure 7.

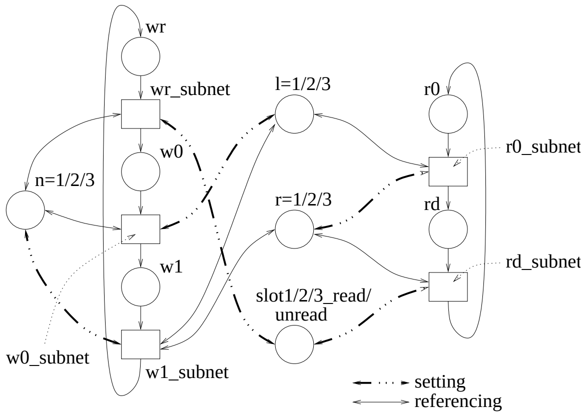

Figure 7: Top level view of 3-slot Pool Petri net model.

In this model, the subnets are virtual transitions which describe the actions of statements in the writer and reader processes in detail. The two FSMs are connected via shared control variables, each of which, as specified in the algorithm, is set by one side only. The other side only reads/references the value of such a variable. In this high-level net, the value of each ternery control variable is represented with a single place. This means that the setting and referencing arcs must be bidirectional to signify that tokens move back and forth between a statement subnet and the modified/referenced control variable during either a setting or referencing statement. Initial condition tokens are not shown in this overview model.

The data items in transit are not represented in the model explicitly, because the analysis process is not interested in any specific data item values being transmitted through the ACM. Only such basic properties as data coherence, data freshness and asynchrony are of interest here. Asynchrony is represented by the two separate FSMs which in the model can progress at their own speed if the Petri net is assumed to follow standard interleaving semantics, to which our analysis conforms. Data coherence and freshness can also be derived from models in the shape of Figure 7 without explicit representation of data items. This will be explained later.

The subnets describing statements can be derived using well-known Petri net representations of value assignment and binary state progression (reading/not reading and writing/not writing of a slot being such states). For instance, the writer statement w1 (n becomes different from l and r) is shown in Figure 8. In this model, "or-arcs" [3, 25] are employed to make the model simpler and easy to read. A group of or-arcs (dashed arcs linked together by a dashed cross-line) form an XOR set linking a transition with more than one place. This signifies that one and only one of the places in question may be marked at any time and the firing of the transition takes a token from this place (and puts it back if the arc is a reference/read arc). Any transition firing in this subnet sets the value of n to be different from the values of l and r according to the lookup table mentioned earlier.

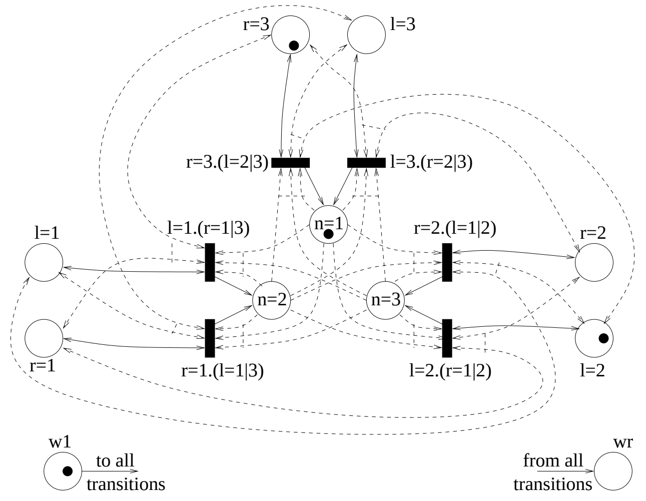

Figure 8: Subnet representing w1: $n := \neg(l, r)$.

The wr and rd statements are represented with non-atomic models with start writing/reading and finish writing/reading being represented by transitions and reading, writing, not reading and not writing of each slot being represented by places. This allows any number of actions in the other side to interleave one data access, illustrating that data accesses are recognized as the major actions that take relatively long periods of time. The models employ broadly similar techniques as those found in Figure 13 below but are simpler because for Pools re-reading is a normal operating mode and need not be caught with monitoring subnets.

The control variable statements w0, w1 and r0 can be represented as atomic, as in Figure 8, or they can be represented as non-atomic [2, 3, 23]. Ideally, for an ACM algorithm to completely conform with specifications in such a form as Figure 5, it should maintain all important temporal and data properties even if all statements are regarded as non-atomic. Representing control variable statements as atomic follows the traditional assumption that the smallest digital action (the setting/reading of a binary or ternery variable) is assumed to be atomic [9, 8, 22]. Representing them as non-atomic has revealed previously unrecognized operating modes [18, 3, 23].

We use monitoring subnets to record such properties as data coherence and data freshness during analysis. Such monitoring nets are used together with the main model of the algorithms in reachability searches to find out the existence of undesirable states or operating modes.

Data coherence is lost iff simultaneous reading and writing occurs on the same data slot. With reading and writing of any slot already explicitely represented by individual places in the wr and rd subnet models, such states can be easily caught using the monitoring subnet shown in Figure 9. If the place "coherence lost" ever gets marked during a reachability search, the algorithm modelled does not maintain data coherence in all operating modes.

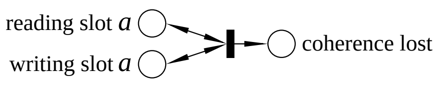

Figure 9: Monitoring data coherence.

Data freshness (as described earlier) refers to the desire that the reader does not obtain stale data. A test for freshness is given in Figure 10 [19]. Figure 10 contains two independent timelines, one each for the writer (A) and reader (B) which should not be read as having any temporal relationship with each other. For the writer timeline, the duration labelled "wr" is the time taken by statement wr and the duration labelled "w0, w1" is the time taken by statements w0 and w1. The arrow pointing to the end of the "wr" duration means that at the end of wr, the slot just written into is set to "fresh". This form of data freshness was discussed in depth in [3] and [23]. Figure 10 shows the addition of the arrow at the start of the read pre-sequence which sets to 'not valid' all slots except the slots which are fresh or previous (if it exists), i.e. the last or previous last one written. As long as the reader obtains the data from these slots or a newer item, freshness is maintained.

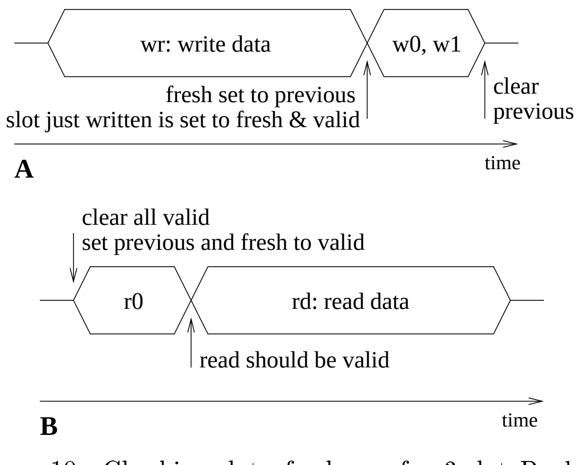

Figure 10: Checking data freshness for 3-slot Pool (A: writer timeline, B: reader timeline).

The subnets modelling the events at the arrows need to be made atomic with the models of their related statements. This can be achieved by a global "enable" place acting like a mutex element in hardware, which maintains relative atomicity of statement and monitoring subnets by enclosing them within critical sections [3, 23].

An overall Petri net model of the three-slot Pool algorithm in Figure 6 was established using the techniques described above. Reachability searches were carried out on this model. The results showed that the algorithm maintains data coherence and data freshness if the statement r0 can be regarded as atomic with regard to the statements w0 and w1 (i.e. the entire sequence of w0 and w1 does not start and finish completely within a single r0 statement in the order of r0 start, w0 and w1 start and finish, r0 finish). If such atomicity cannot be assumed, however, the algorithm may violate both data coherence and data freshness. As an implementation, therefore, it is only correct under certain assumptions of relative atomicity.

In practice, there are many ways in which this assumption of atomicity can be made corresponding to physical situations. For instance, the writer can be made to do other things (such as the preparation of the data item to be written in the next cycle, which should be much more time-consuming than the setting of a control variable) between statements w0 and w1, or arbitration may be employed in hardware to ensure statement atomicity.

6.2 A Signal ACM using two data slots

Because a Signal ACM does not provide its reader with full temporal independence, it is possible to implement it with fewer slots than for a Pool. We again start from a non-atomic model for Signal. This is shown in Figure 11.

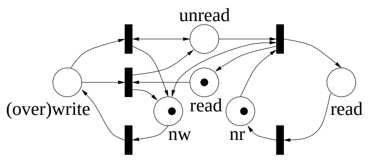

Figure 11: Signal with blocking write and overwrite and non-blocking read.

In this model, it is specified that the writer can initiate and complete a data access at any time, but the reader cannot initiate a data access when a writer access is in progress or when the data state of the ACM is inappropriate (i.e. unread=0). This provides the writer with full temporal independence.

We have developed an algorithm for a two-slot Signal ACM observing these requirements [25]. This is shown in Figure 12.

writer	reader
wr: write slot w	r0: $r := \overline{r}$
w0: $w := \overline{r}$	rd: wait until $w \neq r$
	read slot r

Figure 12: Algorithm of 2-slot Signal.

Here the reader must wait when $w{=}r$. The "wait until" clause specifies that the reader should be stimulated out of waiting by a change of value on w.

The Petri net modelling of this algorithm follows broadly similar lines as in the case of the three-slot Pool.

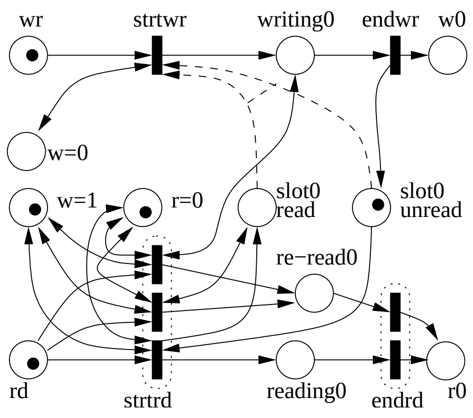

Figure 13: Model of one of the slots of the 2-slot Signal.

In Figure 13, the main model of the read and write data accesses of slot 0 of the 2-slot Signal algorithm is shown. A monitoring subnet for re-reading (re-reading needs to be monitored for Signals) is also included in this diagram. Places “wr”, “w0”, “rd” and “r0” denote the “ready” state for the relevant statements. Transitions with “strt” and “end” in their names represent the start and end of the relevant statements respectively. The place “re-read0” becomes marked if re-reading happens on slot 0, which is how reachability analysis can show up possible faults in the design of the algorithm with regard to the asynchrony requirement of no re-reading.

The top “strtrd” transition in Figure 13 is provided to monitor the cases of potential start reading slot 0 when the writer is in the middle of writing slot 0. This transition is needed because in this situation the main model of the slot shows a data state of neither read nor unread because of the way the writer is modelled. The middle “strtrd” transition in Figure 13 is provided to monitor the cases of potential start reading slot 0 when it has a data state of read. Since all asynchrony and data properties are supposed to be maintained by the control variables, these transitions must be provided to monitor the potential of control variables not working as designed. In other words, if either place “slot0 read” or place “writing0” is ever simultaneously marked with “r=0” and “w=1”, these monitoring subnets will catch it in reachability analysis. If, on the other hand, the algorithm conforms with the specification, these transitions should never fire in reachability analysis. Such transitions are known as “facts”.

The model for read/write accesses of slot 1 is exactly the same shape as Figure 13, with the appropriate label changes. The control variable setting statements are modelled using similar techniques found in Figure 8 and are not shown here.

Reachability analysis has been carried out on the 2-slot Signal algorithm and the results confirm that it satisfies all requirements including asynchronism, data coherence and data freshness. This is true no matter if control variable statements are regarded as atomic or not. This is because there is no loop of control variable dependency in the form of that among n, l, and r in the three-slot Pool. Data freshness monitoring for the 2-slot Signal is according to the timelines shown in Figure 14.

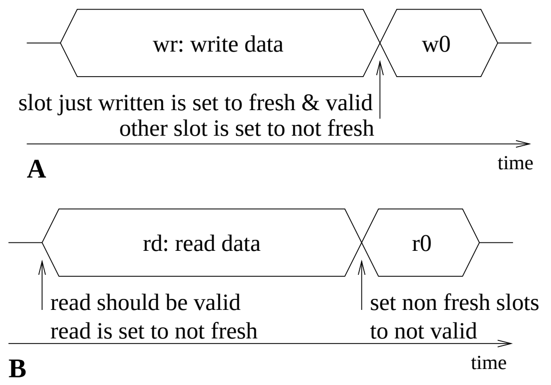

Figure 14: Checking data freshness for 2-slot Signal (A: writer timeline, B: reader timeline).

7 Example ACM hardware designs and simulation results

Both ACM algorithms above have been implemented in hardware. The implementations are based on speed-independent (SI) circuits derived with the direct-translation method based on a type of circuit element known as David cells [15].

As shown above, the 3-slot Pool algorithm in Figure 6 maintains data coherence and data freshness if statement r0 is assumed to be atomic in relation to statements w0 and w1. Because these are control variable statements which should take relatively small amounts of time to execute compared with the data access statements rd and wr, a hardware solution incorporating this atomicity restriction sensibly needs not affect the temporal independence of either writer or reader.

The top-level schematic of a hardware design implementing this algorithm is shown in Figure 15.

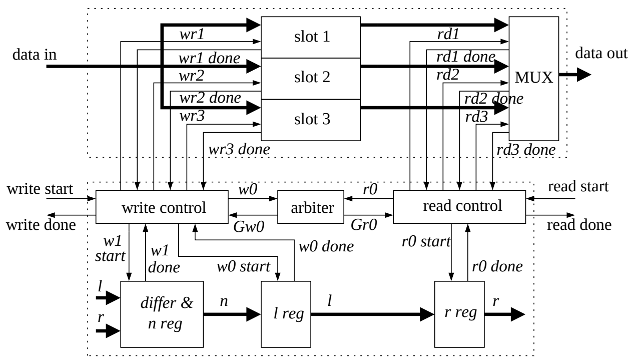

Figure 15: Hardware design of 3-slot Pool.

The lower half of Figure 15 shows the control circuits of the 3-slot Pool. The top half of this figure shows that the values of the control variables are used to multiplex and demultiplex the connections to the three slots from the reader and the writer, so that at each wr and rd statement an access process will be connected to the correct slot. The writer control and reader control boxes contain logic which manages the sequential statements for the write and read sides. The start/done handshakes are used so that the entire control circuit is speed independent (SI) for safety. This includes the completion detection signals being fed back from the data slots to the control circuits. The registers implementing the control variables are built to be SI with full completion detection. The arbiter is used to preserve the atomicity of r0 with regard to w0 and w1.

The circuit implementing w1 using the lookup table approach (the "differ & n reg." block in Figure 15) is shown in Figure 16. It is also entirely SI.

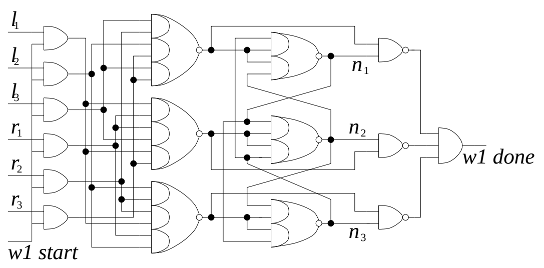

Figure 16: Circuit for w1.

A more detailed description of this hardware implementation of the 3-slot Pool can be found in [24].

For the 2-slot Signal algorithm in Figure 12, a hardware implementation has also been created using similar techniques. Arbitration (via synchronization blocks rather than explicit arbiters) is again used to preserve the atomicity of crucial control variable statements with regard to each other. One point distinguishing Signal ACMs from Pool ACMs is the "wait until" statement in the reader part of the algorithm. While this can be implemented with polling, by using SI circuits it is possible to introduce waiting which does not consume energy. This is more in keeping with the overall view of potentially targetting SoCs as an area of application for ACMs.

The block diagram sketch of the hardware design of the 2-slot Signal ACM based on the algorithm in Figure 12 is shown in Figure 17. Similar to Figure 15, it shows the data path as well as the control parts.

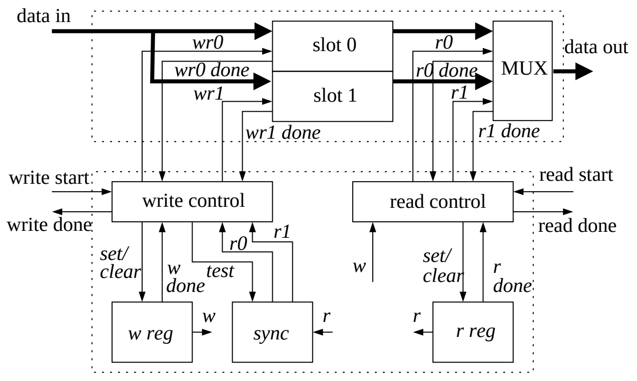

Figure 17: Hardware design of 2-slot Signal.

Again SI circuits have been used for circuit implementation. The write part of the circuit is shown in Figure 18. It consists of a set of David cells [27] (shown 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").

A more detailed description of this circuit can be found in [28].

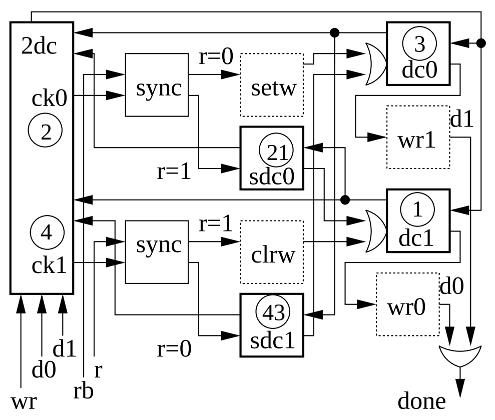

Figure 18: Write part of 2-slot Signal.

The arbiter in Figure 15 and the sync blocks in Figure 18 do introduce possible additional blocking at the level of control variable setting statements that is outside of the specifications. This blocking however will be quantitatively insignificant in operation, assuming that the data access statements are much larger than the control variable ones. If this assumption does not hold, it would be better not to use multi-slot ACMs at all and stick to mutexing on the data access.

Simulation results: The SI nature of the circuit designs ensures that all requirements in the algorithms, such as the sequential order of statements within the reader and writer processes, are satisfied by the hardware. In addition, both analogue and digital simulations were carried out on the circuits implemented with Cadence in VLSI (0.6 micron technology) and on-chip testing was done to another fabricated Pool ACM circuit [28, 14, 24]. These correspond with the results of the theoretical analyses and show that the hardware designs do implement the algorithms.

8 Conclusion and Future Work

An important aspect of the proposed hets, ACMs, has been studied. A classification method for ACMs, focusing on their temporal relationships with their access processes, has been developed. Descriptive definitions of different types of ACMs have been given in an easy to understand formalism, Petri nets. The viability of hets has been demonstrated by the development of algorithmic and hardware implementations of different types of ACMs and their theoretical analyses and simulation.

Het-based systems differ from traditional soft-computing techniques in the following ways: (1) in the treatment of data, errors within data items are not allowed in communications, but items in a sequence can be lost or repeated; (2) in the treatment of time, no global synchrony is assumed and the full range of temporal relations between processes, from fully synchronized to fully temporally independent, is provided. The latter provides for flexible interfaces between elements with self-motivated and reactive timing, potentially facilitating large scale integration without resorting to higher level manipulations of time through such techniques as those found in traditional networks. This has implications in such important considerations as performance and power in SoCs.

We are currently investigating the application of Hets and specifically ACMs in the areas of distributed control systems and distributed vision systems.

9 Acknowledgements

This work has been supported by the EPSRC (UK), grant numbers GR/R32895 at Kingston University and GR/R32666 at Newcastle University, project COHERENT (http://async.org.uk/coherent). Also GR/16754 at Newcastle University, project BESST (http://async.org.uk/besst). The authors wish to thank Alex Bystrov, Eric Campbell, Tony Davies, Hugo Simpson and Walter Vogler for valuable discussions.

References

[1] Burch J., Passerone R., Sangiovanni-Vincentelli A. Overcoming heterophobia: modelling concurrency in heterogeneous systems, Proc. ICACSD2001, IEEE Computer Press, Newcastle-upon-Tyne, UK, June 2001.

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

[3] Clark I.G., A unified approach to the study of asynchronous communication mechanisms in real time systems, Ph.D. Thesis, London University, King's College, May 2000. (http://IanGClark.net/)

[4] Cravotta R. Exploring the anatomy of Multiprocessor Designs, EDN Europe, November 2002, pp.49-60.

[5] Dettmer R. Where next for the microprocessor?, IEE Review, November 2002, p7.

[6] Horstmann J.U., Eichel, H.W. and Coates, R.L. Metastability behaviour of CMOS ASIC flip flops in theory and test, IEEE Journal of Solid State Circuits, Vol. 24, No. 1, pp.146-157, February 1989.

[7] Kinniment D.J., Chester E.G. Design of an On-Chip Random Number Generator using Metastability, Proc. of the 28th European Solid-State Circuits Conference (ESSCIRC) 2002, 24-26 Sept. 2002, Florence, Italy.

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

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

[10] Madalinski A., Xia F., Yakovlev A. Studying the data loss and data re-reading behavious of a four slot ACM using SPN techniques, Proc. of 7th UK Asynchronous Forum, 20-21st December 1999, University of Newcastle-upon-Tyne.

[11] Marino L.R. General theory of metastable operation, IEEE Trans. Comput., Vol. 30, No. 2, pp.107-115, 1981.

[12] The Moses Project, Modeling, Simulation, and Evaluation of Systems: http://www.tik.ee.ethz.ch/˜moses/.

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

[14] Shang D., Xia F., Yakovlev A., Testing a self-timed asynchronous communication mechanism (ACM) VLSI chip, IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS) 2001, Gyor, Hungary, 18-20 April 2001. pp. 53-56.

[15] Shang D., Xia F., Yakovlev A., Asynchronous circuit synthesis via direct translation, Proc. ISCAS 2002, Phoenix, Arizona, May 2002.

[16] Shang D. Asynchronous Communication Circuits: Design, Test and Synthesis. PhD Thesis, University of Newcastle, September 2002.

[17] Simpson H.R. The MASCOT method, Software Engineering Journal, Vol.1, No.3, pp. 103-120, 1986.

[18] Simpson H.R. Four-slot fully asynchronous communication mechanism, IEE Proceedings, Vol. 137, Pt. E, No. 1, pp.17-30, January 1990.

[19] Simpson H.R. Correctness analysis of class of asynchronous communication mechanisms, IEE Proceedings, Vol. 139, Pt. E, No. 1, pp.35-49, January 1992.

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

[21] 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.

[22] Tromp J., How to construct an atomic variable, Proc. 3rd Int. Workshop on Distributed Algorithms, Nice, LNCS, Spring Verlag, pp.1.0-302, 1989.

[23] Xia F., Supporting the MASCOT method with Petri net techniques for real-time systems developement, Ph.D. Thesis, London University, King's College, January 2000.

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

[25] Xia F., Clark I., Algorithms for Signal and Message Asynchronous Communication Mechanisms and Their Analysis, Proc. ICACSD2001, IEEE Computer Press, Newcastle upon Tyne, UK, June 2001.

[26] Xia F., Yakovelv A.V., Clark I.G., Shang D. Data communication in systems with hetergeneous timing, IEEE Micro, Nov-Dec 2002.

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

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