Data Communication in Systems with Heterogeneous Timing

Fei Xia, Alex V. Yakovlev, Ian G. Clark*, Delong Shang
Department of Computing Science, University of Newcastle upon Tyne, UK

{fei.xia; alex.yakovlev; i.g.clark; delong.shang@ncl.ac.uk}

Abstract

This paper discusses asynchronous communication mechanisms (ACMs). A system of classifying ACMs and example hardware implementations of ACMs are presented. ACMs can be very useful in computational systems with inherent heterogeneous timing, especially in real-time, embedded applications.

1 Introduction

Motivation: In recent years, such "soft-computing" technologies as fuzzy logic, genetic algorithms and neural networks have become increasingly popular, particularly in the context of massively parallel computational environments. The unifying characteristic of these technologies is their somewhat relaxed view of the importance 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 unviolatable, 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. However, global synchrony is undesirable for distributed real-time systems and is also becoming increasingly impractical with the rapid increase of clock frequencies.

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 or making Systems-on-Chip (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.

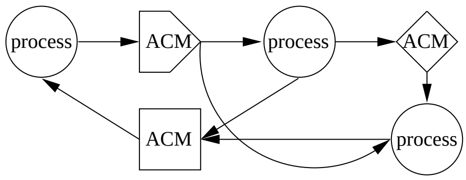

Figure 1: A het is a digital network with computational elements connected through ACMs.

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

*also with School of Computing and Information Systems, Kingston University, UK the system assembly from sub-systems and elements.

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 and hardware design.

Previous work: The MASCOT (Modular Approach to Software Construction Operation and Test) design methodology, developed primarily at BAe and RSRE [1] for constructing real-time distributed systems of a safety-critical nature, provides initial stimulus for studying systems with multiple active timing domains (sometimes called "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 (there is much more custom digital electronics in most missiles) 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).

2 ACMs and their properties

ACMs in the literature: 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 Figure 2.

Figure 2: Passing data via shared memory.

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 further assumed that the data being passed consists of a stream of individual items of a given type. It is also assumed that the processes in question are single thread cycles, one providing and the other making use of a single item of data during each cycle. The provider of data is known as the "writer" and the user of data is known as the "reader" of the ACM.

Lamport classified ACMs into "safe", "regular" and "atomic" "registers" [3] and tried to assemble the more useful regular and atomic types out of directly realizable safe types. Exploiting the fact that the smallest such "registers" (ACMs) that transmit bit type data items can be implemented to atomic standards directly, many ACM protocols have been proposed in the literature (e.g. [4, 5, 6]) which employ less than safe registers for the data path whose access by the reader and writer processes is managed by minimal (bit) control variables. The results of these efforts are much better and more practical ACMs performing to atomic or regular standards. All these ACMs allow full temporal independence between the reader and writer processes.

Simpson then proposed a more general classification system for ACMs, incorporating protocols that do not necessarily allow full temporal independence for reader and writer [7, 8].

Asynchrony and its consequences: overwriting and re-reading: It is important to classify ACM protocols based on whether the data state of an ACM may oblige either the reader or writer processes to wait under certain circumstances. 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 is 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 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 contained in the ACM, should normally be regarded as more important (cf. freshness property defined in Section 2). 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 new data items ready to be read, 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 either the reader or 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.

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 Table 1.

In Table 1, 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 [8]. A new ACM type, "Message", is introduced as the dual of Signal.

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

	NRR	RR
NOW	Channel	Message
OW	Signal	Pool

In terms of the data state blocking data access, 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.

Most published ACMs have non-zero capacities. Zero capacity ACMs have also been mentioned [8].

Most 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[1]. 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.

ACM symbols and combinations: We propose to use a set of unique symbols for ACMs so that an architecture view of a het with ACMs may be presented graphically. For this purpose, the temporal relation between an access process and an ACM must be specified by such symbols.

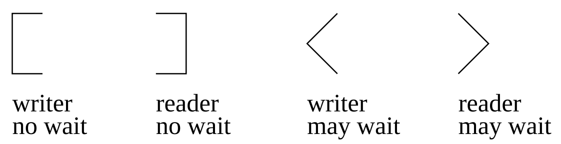

Figure 3: Symbols for temporal relations between reader/writer and ACM.

[1]Incidentally, the notion of dependability is normally more general than fault tolerance and unduly neglecting the temporal aspect of it, such as the need and provisions for asynchronous interactions, may lead to catastrophic consequences.

Symbol elements specifying whether an access process may be required to wait at an ACM are given in Figure 3. With these elements, it is possible to construct graphical symbols for each of the four ACM types in Table 1. These are shown in Figure 4.

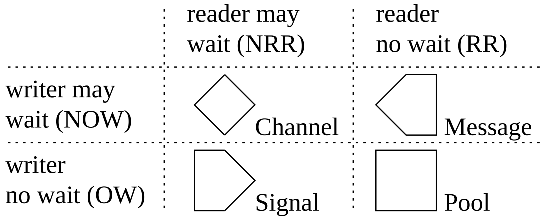

Figure 4: Graphic symbols for ACM types.

With these graphic symbols, it is straightforward to assemble an architecture view of any het with active elements connected to one another through ACMs. In addition, the notion of directly combining a number of ACMs into a composite one can be easily expressed. For instance, if a buffer of size n (an n-capacity Channel) is connected to a Signal on the input side and a Message on the output side, the resulting composite ACM would look like what is shown in Figure 5.

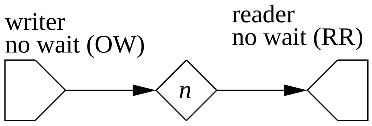

Figure 5: A composite Pool type ACM with n-capacity buffer inside.

This composite ACM would behave to its reader and writer like a Pool in terms of whether overwriting or re-reading are permitted. However, it would also have an internal buffer which may help "smooth" the rate of flow of data items. The non-blocking FIFO introduced in [10] is a non-modular implementation of this ACM.

Petri net models of reading and writing: In order to describe the relations of asynchrony, data states, reading and writing in the context of ACMs, Petri net [11] models have been developed. 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 analysis at a level where the detailed implementations of ACMs are not important.

For simplicity, all models in this section assume that the relevant ACMs have data capacity 1. In other words, an ACM contains one item of data at all times. This does not represent a loss of generality. Models introduced here can be easily extended to cover non-unity capacity cases. Examples of these extended models can be found in [14].

If the write and read data accesses are regarded as atomic processes, their modification of the data state can be represented by the models in Figure 6.

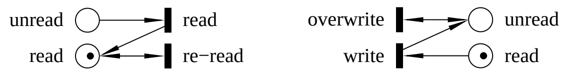

Figure 6: Atomic reading and writing.

Writing and reading, however, are generally processes that take time, and it is often unrealistic, especially in real-time systems, to assume that other processes may not start or finish when one of these data access processes are taking place.

Apart from the asynchrony issues relating the data state to an access process, the same issues exist between the two access processes of any ACM. For instance, the data state may not cause waiting for the writer, but the implementation may be of such a nature that read accesses may. In other words, the writer may be allowed to start and complete an access no matter what the data state is but in the middle of a read access the writer is not permitted to start an access of the ACM.

It is, therefore, often important to represent the write and read accesses to an ACM as non-atomic. For the simple P/T nets (place/transition or "classical" Petri nets) used here, this can be done with the technique proposed in [12].

P/T net fragments modelling the access processes as non-atomic and specifying the blocking/non-blocking nature of these processes to their opposite numbers have been developed and presented below.

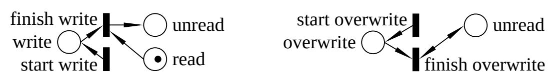

Figure 7: Non-atomic writing and overwriting.

Figure 7 shows the basic models of writing and overwriting as non-atomic actions. The data state is modified at the end of writing or overwriting, indicating that only at the completion of a write/overwrite access a new unread item of data may be regarded as having appeared in the ACM.

Figure 8 shows the basic models of reading and re-reading as non-atomic actions. The data state is modified at the start of reading and re-reading. This means that once a read access starts the item of data being

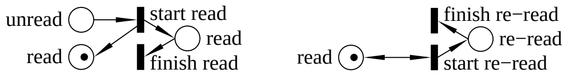

Figure 8: Non-atomic reading and re-reading.

accessed is regarded as having been read. Considering the original assumption that the reader obtains one item of data in each cycle through a single read data access to the ACM, this causes no controversy if "retry loops" within individual read accesses are not permitted. Retry loops, whether within data accesses or otherwise in the individual cycles of reader and writer threads, have been regarded as undesirable for many reasons [13]. In this paper all ACMs are defined as without such loops.

The models in Figures 7 and 8 are adequate for cases where read or write accesses do not block accesses of the other side. If accesses by one side block those by the other, however, complementary places to the writing and reading ones must be used to represent such mutual exclusion completely.

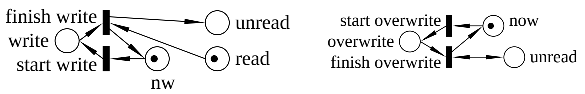

Figure 9: Non-atomic writing and overwriting with complementary places.

Figure 9 has complementary places whose marking indicates the states of not writing or not overwriting. The inclustion of similar places can be added to the read access models in Figure 8. These places will enable the modelling of blocking by the data access of one side of the data access of the other side without resorting to such devices as inhibitor arcs [11].

Additional properties of ACMs: 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 sequentiality, data loss and data repetition. Other properties, such as power and hardware efficiency and temporal consistency, have not been dealt with in detail.

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.

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 once.

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.

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. 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.

3 Descriptive models of ACMs

In this section the various ACM types are described with Petri net models which are the basic state-transition definitions of these protocols.

In complex systems, the temporal requirements of different processes may differ considerably. Some processes may exist in a very tight temporal regime for real-time and/or safety critical reasons. For such processes, it is important to be able to predict their progress with relation to time. This means that they are often mapped to hardware controlled by predictable, often rigid, clock signals. Other processes may exist in a fairly relaxed regime with respect to timing. For such processes concerns of maximising efficiency, whether of power or of silicon, may outweigh predictability of their temporal behaviour with regard to real-time.

In many applications, there may be a need for processes existing in real-time and flexible time domains to communicate by passing data. This is where the Signal and Message type ACMs may be most relevant.

The basic definition of Signal treating read and write accesses as atomic is shown in Figure 10. 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.

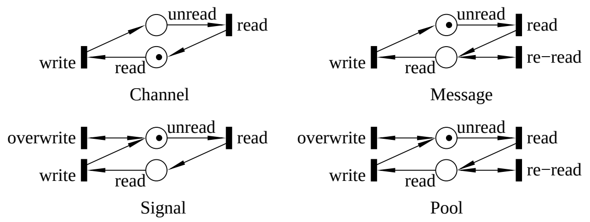

Figure 10: ACMs with atomic reading and writing.

It is obvious from this definition that Signal is most useful in connecting a writer process which exists in a relatively tight temporal regime with a reader process which exists in a relatively flexible temporal regime. 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.

On the same basic level, treating read and write accesses as atomic, the Message protocol is also defined in Figure 10. This is a perfect mirror 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 a relatively tight temporal regime with a writer process which exists in a relatively flexible temporal regime. 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, the Pool type ACMs may be specified between writer and reader processes both living in more rigid timing domains or when it is otherwise a good idea to not allow data communications to affect the timing of either communicating side, while the Channel type ACMs may be specified when the integrety of the data item stream being transmitted is paramount and the timing requirements of both reading and writing processes can be compromised to accomodate this. The Petri net definitions of both Channel and Pool can also be found in Figure 10.

4 Example ACM hardware designs

We have designed Pool and Signal ACMs in hardware in a number of different ways. Here two examples are presented to illustrate the common problems and techniques in ACM hardware design.

A Pool ACM using three data slots: The models in Figure 10 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 in most applications when 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 11. This specification gives total temporal independence to both reader and writer with regard to the ACM and each other.

The most direct method of implementing such a Pool specification is to restrict any timing conflicts between the two sides to signals of the smallest granularity within the system's scope. In most cases this means binary or ternary variables whose setting and resetting can be accomplished in hardware within the smallest possible action time. It is possible, by employing such "control variables", to remove any asynchrony from more large-scale actions such as the accessing of a data item by one of the sides.

In the terminology of multi-slot ACMs [6], a "data slot" is a unique portion of the shared memory which may contain one item of data. A Pool specified by Figure 11 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.

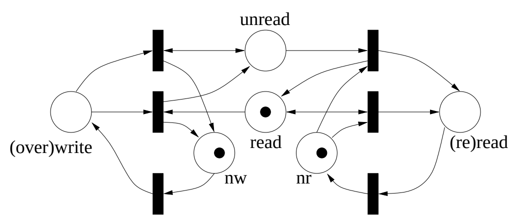

Figure 11: Non-atomic Pool model

It has been shown [6, 20, 14] that a fully asynchronous Pool can only be implemented using three or more data slots. Here we present a hardware solution using three data slots, based on the algorithm found in [6] shown here in Figure 12.

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

Figure 12: Algorithm of 3-slot Pool.

In this algorithm, 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, the statements must be executed in the order specified and any statement may not start without the previous one having completed. This algorithm has been shown to maintain full data coherence and data freshness if statement r0 is assumed to be atomic in relation to statements w0 and w1 [14]. 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 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 13.

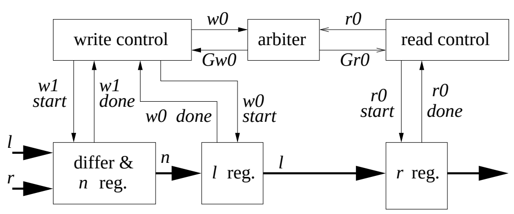

Figure 13: Hardware design of 3-slot Pool.

Figure 13 shows the control circuits of the 3-slot Pool but omits the data slots. It is assumed that the values of the control variables will be 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. 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 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)). The circuit implementing w1 (the "differ & n reg." block in Figure 13) is shown in Figure 14.

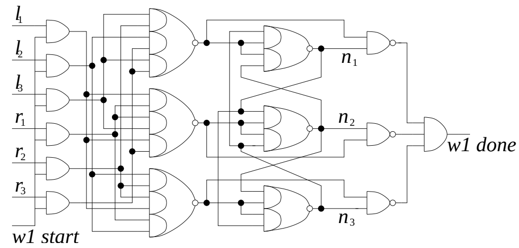

Figure 14: Circuit for w1.

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 15.

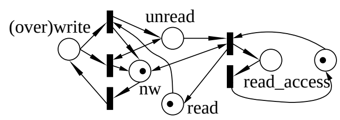

Figure 15: Signal with blocking write and overwrite and non-blocking read.

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 not correct (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 [21]. This is shown in Figure 16.

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

Figure 16: 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. While this can be implemented with polling, it is best done in hardware using SI circuits where waiting does not consume energy.

The block diagram sketch of the hardware design of a 2-slot Signal ACM based on the algorithm above is shown in Figure 17. At a higher level than Figure 13, it shows the data path as well as the control parts.

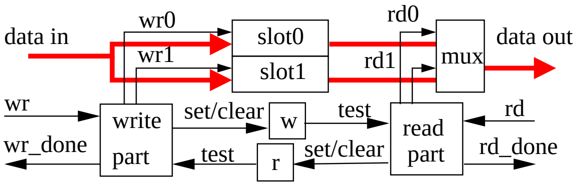

Figure 17: Hardware design of 2-slot Signal.

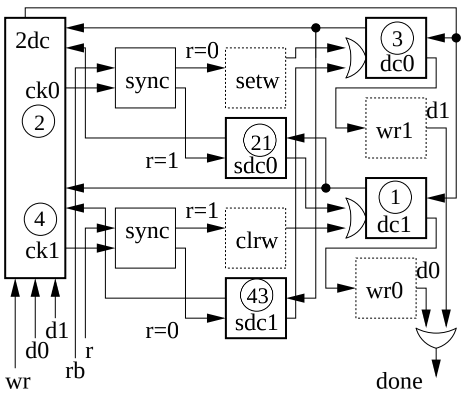

Figure 18: Write part of 2-slot Signal.

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 [18] (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").

A more detailed description of this circuit can be found in [19].

The arbiter in Figure 13 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: Theoretical analyses of these Pool and Signal algorithms have been published [21, 20, 14], and the SI nature of the circuit designs ensure 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, simulations were carried out on the circuits implemented with Cadence in VLSI and on-chip testing was done to a fabricated Pool ACM circuit [19, 22, 9]. These support the results of the theoretical analyses and show that the hardware works to specifications.

One series of the simulation results illustrate the interesting aspect of trading data items for temporal independence. This is shown in Figures 19 and 20.

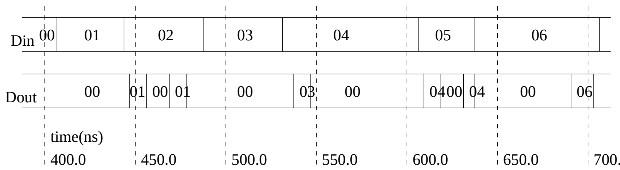

Figure 19: Value sequences of input and output data at the beginning of a simulation.

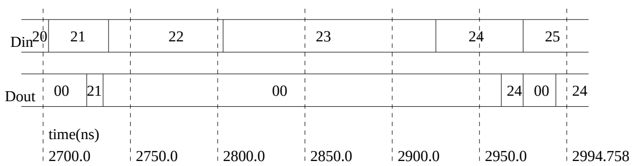

Figure 20: Value sequences of input and output data in the middle of a simulation.

This simulation was run on the Pool ACM circuit and the figures show the values of data items coming into and being extracted from the Pool by the writer and reader respectively. The "00" parts are spacer states on the reader's side when the reader is not carrying out a data access. Random number generators are used to vary the duration of reader and writer cycles dynamically. It should be noted that both data freshness and data coherence are maintained even under such variable operating modes. Overwriting and re-reading are clearly shown in the sequences, but other recorded data also show that there is no ACM-imposed delays on either side. So full temporal independence is provided for both reader and writer.

5 Conclusion

An important aspect of the proposed hets, ACMs, has been systematically studied. A new 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 hardware implementations of different types of ACMs.

Other related work, including the systematic synthesis of ACMs [19] and the development of automatic verification techniques for ACMs [21], as well as quantitative analyses of such properties as data loss and data re-reading in ACMs [23], also strengthen the foundation for the het-based methodology of real-time and embedded systems.

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.

References

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

[2] Yakovlev, A., Nouta, R. (Eds.) Proceedings of AINT'2000 (Workshop on Asynchronous Interfaces: Tools, Techniques and Implementations), Delft, July 2000.

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

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

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

[8] 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.

[9] 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.

[10] Yakovlev, A., Kinniment, D.J., Xia, F., Koelmans, A.M., "A FIFO buffer with non-blocking interface", IEEE Computer Society TCVLSI Technical Bulletin, Fall 1998, pp. 11-14.

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

[12] 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.

[13] 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, IEEE Computer Society, Berlin, Germany, June 17-19, 1998.

[14] 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.

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

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

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

[18] 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.

[19] 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.

[20] 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.

[21] Xia, F., Clark, I., "Algorithms for Signal and Message Asynchronous Communication Mechanisms and Their Analysis", Fundamenta Informaticae, Vol.50, 2002 (to appear).

[22] 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.

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