Data Communication in Systems with Heterogeneous Timing

Asynchronous communication mechanisms permit the implementation of data interfaces between heterogeneously timed entities at various hardware levels. A systematic approach to ACM classification, specification, and implementation facilitates their use in heterogeneously timed networks.

Fei Xia
Alex V. Yakovlev
University of Newcastle upon Tyne

Ian G. Clark
University of Newcastle upon Tyne and Kingston University

Delong Shang
University of Newcastle upon Tyne

Practical examples of systems with heterogeneous timing requirements include the embedded real-time control and decision-making systems used in aerospace, automotive, and telecommunication applications. It is attractive to construct such systems as digital networks with inherent heterogeneous timing conditions. In these *hets* (short for heterogeneously timed nets), computational elements interact through asynchronous communication mechanisms (ACMs). Hets offer applications

- a more formal recognition of system timing heterogeneity,
- a networked and potentially hierarchical view of the system composed of clearly active and passive elements, and
- direct representation of data flow through the system.

These characteristics facilitate both top-down and bottom-up design approaches and the assembly of systems from subsystems and elements.

Why not synchronous?

There are several reasons why distributed-system subsystems cannot all belong to the same timing domain. Some of these reasons have to do with power considerations. For instance, subsystems can have bounded energy resources; having such subsystems enter sleep mode might be desirable if there is no change (or need for change) in their input/output data. It's also possible for subsystems to have operational speed depend on local energy supply conditions—that is, a subsystem can slow down when its energy reserve becomes low—rather than have operational speed depend on communications with other subsystems.

From the application point of view, many applications (such as portable equipment, and control or signal processing systems) must guarantee real-time regimes in some of their parts and power savings in others. From the system implementation point of view, the *International Technology Roadmap for Semiconductors* (http://public.itrs.net) indicates that systems on a chip (SOCs) are increasingly becoming heterogeneous in behavior,

0272-1732/02/$17.00 © 2002 IEEE

because they now include mixed analog-discrete components, and time-driven and power-saving subsystems. New system designs are also becoming increasingly communication centric. These factors create an urgent need for data interfaces between subsystems and processes with different temporal and power requirements.

Such interfaces must emphasize real-time and energy consumption characteristics. They must also adjust to the inherent contradiction between the desired temporal and power characteristics of client processes, requiring radical changes at the level of data communication mechanisms. These changes include the use of asynchronous or clock-free circuits and more direct use of hardware instead of software layers to support communication protocols. (Hardware provides better temporal predictability and power saving than software.)

In addition, 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 an individual data item's precision. The increased robustness of such techniques makes it possible to soften data precision requirements, leading to various implementation advantages.

Similar levels of attention, however, have not been paid to the possible softening of requirements for temporal relations among processing elements in complex systems. Even in soft-computing technologies, designers have almost invariably assumed that systems operate under global synchrony. In addition, established soft-computing technologies incorporate data softening within processing units, but assume conventional, full-precision data communications among processing units. In other words, such designs still require communication units to treat data passing through them as inviolable, although, from the systemwide point of view, the precision of individual data items is unimportant. Such a dichotomy does not present a problem if designers assume that systems operate under global synchrony. However, global synchrony is undesirable for distributed real-time systems and is also becoming increasingly impractical with the rapid increase in clock frequency and integration level for single chips.

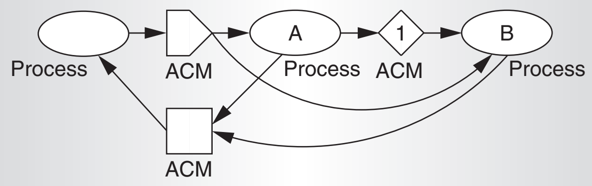


Figure 1. A het is a digital network with computational elements connected through ACMs.

Heterogeneous timing

Alternatively, a system might require

- every data item generated by a processing unit to reach its destination, and
- the intended receiver of a data stream receives an exact copy of the stream generated for it without any error.

Such requirements still must have some synchronization between the generating and receiving processes, which in turn implies that all system processing units need to temporally relate to one another. Although considerably less restrictive than global synchrony or a single clock, these requirements might still be 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 have temporal independence for the purpose of real-time safety, power savings, or making SOC implementations more practical. In other words, softening synchrony by permitting data loss or repetition potentially caters to wait-free operations for components and helps avoid the need for unified clocks. It therefore holds attractions for both real-time operations and very-high-scale system integration.

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

Asynchronous communication mechanisms

Heterogeneously timed systems might need to maintain data interfaces between subsystems in different timing domains. The minimal form of this problem is the unidirectional passing of data between two single-thread processes.

When the two communicating processes

are unsynchronized, it's often necessary to pass the data through some intermediate data repository, such as in the path formed by processes A and B, and ACM 1 in Figure 1.

The ACM scheme manages data transfer between two or more processes not necessarily synchronized for the purpose of data transfer. Here, we assume the involvement of only two processes and that the data is a stream of individual items of a given type. We further assume that the processes in question are single-thread cycles, one providing and the other using a single item of data during each ACM access. The data provider is the ACM *writer*; the data user is the *reader*.

Lamport classified ACMs as *safe*, *regular*, and *atomic* registers[1] and tried to assemble the more useful regular and atomic types out of the more directly realizable safe type. The smallest such "registers" transmit data items of the single-bit type. Ignoring nondigital behavior such as metastability, these registers are implementable to atomic standards. Exploiting this fact, researchers have proposed many ACM protocols.[2] These protocols typically employ less-than-safe registers for the data path, and minimal (bit) control variables manage access by reader and writer processes to the data path. The results of these efforts are better and more practical ACMs that perform to atomic or regular standards, assuming atomic standards in the control variables. All these ACMs allow full temporal independence between reader and writer processes.

Simpson proposed a more general classification system for ACMs, incorporating protocols that do not necessarily allow full temporal independence for reader and writer.[3] Compared with Lamport's work, which emphasizes data, Simpson's classification moves emphasis to the temporal relations between reader and writer processes. Here, we further develop this classification by focusing directly on the temporal relations between ACMs and their access processes.

The literature has consistently yielded research that relaxes synchronization requirements from an assumption of full data transmission. This research includes globally asynchronous, locally synchronous systems (GALSs),[4] stretchable/pausible clocks,[5] FIFO buffers allowing more temporal differences,[6] and most notably, in recent years, latency-insensitive design.[7] Because of the starting point of full data transmission, none of these provide a full spectrum of temporal relations between the communicating sides. Such a spectrum would run from fully independent to fully synchronized and operate at the lowest hardware level.

Our ACM approach starts from the temporal relations between the reader and writer, and regards data transmission quality to be of secondary importance. This view differs radically from the traditional data-centric approach to data communications, and makes possible true temporal decoupling at the hardware level.

Asynchrony and its consequences

From the hets point of view, it's important to classify ACM protocols based on whether an ACM's data state might oblige either the reader or writer processes to wait under certain circumstances. Without going into the details of specification and implementation, we can classify ACMs into four groups based on the combination of might- or might-not-require waiting for the two access processes.

The implied desire for any ACM is that as much of the writer's information should pass to the reader as possible, once the data transfer satisfies the ACM's basic asynchrony specifications. This means that when a no-waiting requirement implies imperfection in data transfer, designers should find a method to minimize such imperfection.

There is also an implied desire for any ACM to accord as much asynchrony as possible to the writer and reader, while staying within the specification. This means that an ACM must invoke any waiting in a might-require-waiting specification only when absolutely necessary.

For the writer, therefore, the question of waiting or not waiting comes up only when an ACM is full of unread data items. If it's important that every data item from the writer must eventually reach the reader, the ACM must require the writer to wait in this situation. On the other hand, if the ACM cannot oblige the writer process to wait for data communication purposes, it must discard or lose data. Straight discarding of the current data item is often not a good solution because a newer data item should normally be more important (per the freshness property we define later) than those already in the ACM. It's, therefore, often better to overwrite data; that is, the current data item the writer has ready to write should replace some item already in the ACM.

Similarly, when an ACM contains no previously unread data items, to prevent the data state from blocking the reader process, the ACM must let the reader go away without reading or read some data item it had already acquired during a previous access. The best option here to ensure some data passage with this reader visit is to let the reader reread the data item from the previous data access.

Overwriting and rereading are also superior to straight discarding or no reading because they take essentially the same time (in reasonable implementations) as normal writing and reading. This way, a temporal consistency exists for ACM accesses by the reader and writer. Such temporal consistency is crucial—or at least desirable—in many hard, real-time, safety-critical systems.

Permitting overwriting but not necessarily requiring it at all times implies that the data state cannot hold up (block) the writer. Permitting rereading but not necessarily requiring it at all times implies that the data state cannot hold up (block) the reader. Our new ACM classification system categorizes these protocols based on whether an ACM protocol permits these actions. This classification system fully corresponds to the no-waiting-required/waiting-might-be-required paradigm.

Classifying ACMs

Formally, an ACM has a *capacity*, the number of data items it contains; this is a nonnegative integer constant. At any time, each data item an ACM contains is either read or unread. An ACM's basic data state consists of its number of unread data items.

We divide write data accesses into writing and overwriting. Read data accesses are either for reading or rereading. 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); overwriting and rereading do not modify the data state. Overwriting, if permitted by the ACM protocol, can only occur when the ACM's data state equals its capacity—that is, all of its data items are unread. Rereading, if permitted by the ACM protocol, can only occur when the ACM's data state is 0—none of its data items are unread.

We classify ACMs according to whether they permit overwriting and rereading. The Channel, Pool, and Signal ACM protocols, which we discuss later, inherit their names from an earlier classification scheme.[3] We introduce a new ACM type, called Message, as the dual of Signal.

In terms of the data state's blocking of data access, if an ACM protocol permits rereading, it does not hold up the reader. If a protocol permits overwriting, it does not hold up the writer. If a protocol does not permit rereading, the reader must wait when the data state is 0. If the protocol does not permit overwriting, the writer must wait when the data state equals the ACM's capacity.

Traditional computer systems, if they do not interface to analog or real-time environments, will only use Channel protocols—in the form of FIFO, LIFO, or RAM buffers—for communication. This is because for these systems, asynchrony is secondary to data preservation. These solutions don't permit data loss and repetition so must maintain some synchronization. Typically, such systems treat data loss and repetition as anomalies, and these situations are therefore part of fault tolerance considerations, which are well studied in hardware implementations. (Incidentally, the notion of dependability is normally more general than fault tolerance. Unduly neglecting dependability's temporal aspects, such as the need to support asynchronous interactions and provisioning for that support, can lead to catastrophic consequences.)

For these reasons, we focus attention on the Signal and Pool protocols. Because Message is Signal's dual, any work on Signal also sheds light on Message.

ACM symbols and combinations

We propose to use a set of unique symbols for ACMs to graphically represent an architectural view of a het with ACMs. Such symbols specify the temporal relation between an access process and an ACM. Figure 2 shows symbol elements that specify whether

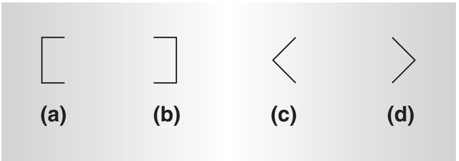
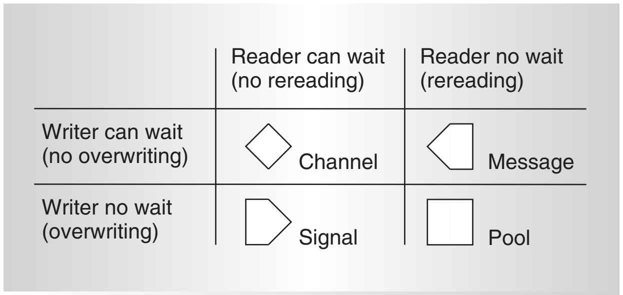


Figure 2. Symbols for temporal relations between the reader or writer and ACM: writer (a) and reader (b) no wait; and writer (c) and reader (d) may wait.



	Reader can wait (no rereading)	Reader no wait (rereading)
Writer can wait (no overwriting)	Channel	Message
Writer no wait (overwriting)	Signal	Pool

Figure 3. ACM classification and graphical symbols for ACM types.

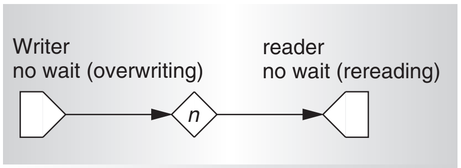


Figure 4. Composite Pool with *n*-capacity buffer inside.

a protocol can require an access process to wait at an ACM. We combine these symbols to form other symbols, shown in Figure 3, to represent the four ACM types.

With these symbols, it's straightforward to assemble an architectural view of any het with active elements connected to one another through ACMs. Figure 1 is one example. In addition, it's easy to express the notion of directly combining several ACMs into a composite. For instance, if a buffer of size *n* (an *n*-capacity Channel) connects to a Signal on the input side and a Message on the output side, the resulting composite ACM would look like Figure 4 and behave like a Pool to its reader and writer in terms of overwriting and rereading permission. The internal buffer can help the flow rate of data items stay more even than that of a simple Pool, at the expense of extra latency.

Practical significance of various ACMs

Systems with heterogeneous timing can incorporate

- active elements with self-motivated timing (these elements internally determine their own timing), and
- other active elements that have reactive timing, that is, their timing depends on their data communication interfaces.

More complex situations, such as a single active element having a combination of self-motivated and reactive timing, can also exist.

For any specific ACM, its reader and writer processes can belong to active elements that might need self-motivated or reactive timing with respect to this ACM. Self-motivated timing is most useful when an active element must

- perform real-time tasks;
- work in a system where precise timing and temporal predictability are important; or
- have specific, controllable power and/or performance requirements and speed up or slow down without considering its relationship with an ACM.

On the other hand, reactive timing can be useful when an active element performs tasks with enough temporal flexibility to accommodate waiting at an ACM. Such waiting might be necessary to reduce data loss and/or repetition, or to reduce power consumption—that is, the active element does nothing when no new data is available or needed.

In general, not requiring an access process to wait lets that process belong to an active element with self-motivated timing. On the other hand, requiring an access process to wait occasionally implies some degree of reactive timing in the active element that incorporates the access process. Each of these situations benefits from the use of a particular type of ACM:

- Pools can interface two elements with self-motivated timing.
- Signals and Messages are best for interfacing elements with self-motivated timing to elements with reactive timing.
- Channels can interface two elements with reactive timing.

ACMs make possible real asynchrony between active elements in a system and do so at the lowest hardware level. This capability has significant implications in regulating and saving power. For instance, it's possible to implement Pools so that both sides enjoy full temporal independence from each other. Designers can use this type of full temporal decoupling to localize any temporal fluctuations caused by dynamic power control in parts of a system.

Designers can implement Signal and Message ACMs so that communicating element 1 can have complete temporal independence from element 2, but element 2 is not independent of element 1. Such ACMs are useful when communication affects the timing of element 2 (say to save power), but doesn't affect element 1's timing. For instance, hardware implementing a reader could be off when no new data is available, or hardware implementing a writer could be off when the reader doesn't need new data. Conventional communication schemes don't easily allow this type of operation, at least not at the hardware's finest granularity. With appropriate hardware implementations of ACMs and active elements, such turning off and subsequent restarting can be low energy events.

Additional ACM properties

In addition to asynchrony, ACMs have other important properties investigated in previous work. An ACM's main data-passing properties include data loss, repetition, coherence, and freshness. Other properties, such as power and hardware efficiency, and temporal consistency are also important. Research has not dealt with some of these areas in detail.

Data loss and data repetition are the inevitable consequences of no waiting and ACM implementations of bounded size. Data loss occurs because of overwriting; one data item previously introduced by the writer becomes permanently unavailable to the reader—in other words, lost in transit. Data repetition occurs because of rereading. In this case, the reader obtains a data item it obtained in an earlier cycle.

Data coherence refers to the integrity of individual data items that pass through an ACM. This means that data items obtained by the reader should remain unmodified after their introduction by the writer. In other words, data items going through an ACM should retain their individual integrity or atomicity. If a data item changes between the writer and reader, data coherence is lost.

Data freshness refers to the reader obtaining the most up-to-date data item in an ACM that satisfies the particular protocol. For instance, with a Pool of capacity 1, any read should obtain the data item introduced by the last write that completed before it.

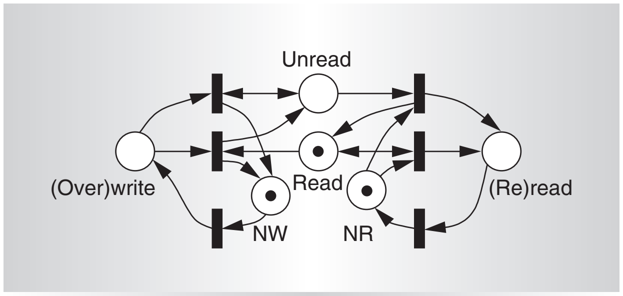


Figure 5. Nonatomic Pool model.

Power efficiency refers to minimizing the power consumption of ACM hardware implementations. Hardware efficiency refers to implementing any ACM with the smallest possible silicon area. Depending on the implementation, these two properties can be related.

Temporal consistency refers to having uniform and predictable temporal characteristics for a given writer or reader. For example, all read accesses to an ACM from the same reader should require the same predictable amount of time to complete. Other well-known properties, such as latency and throughput, might also affect ACM design.

Algorithm designs and analyses

We have designed several Pool and Signal algorithms; two examples illustrate our method of ACM analysis and verification.

A Pool ACM with three data slots

The Petri net[8] fragment in Figure 5 specifies a Pool. In this specification, two complementary places—*unread* and *read*—represent the data state of the Pool. Two other complementary places—*(over)write* and *nw*, represent the state of the writer. Place (over)write is marked when the writer is in a data access of the ACM, and place nw is marked when the writer is not in a data access of the ACM.

Similarly, the complementary pair of places *(re)read* and *nr* represents the reader state. Double-headed arcs are *contextual* arcs, indicating that although the transition is enabled only when the place is marked; the transition's firing does not consume a token from the place. The data state is modified at the end of a write access (if writing is incomplete, an item is not ready to be read) and the beginning of a

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

Figure 6. Three-slot Pool algorithm.

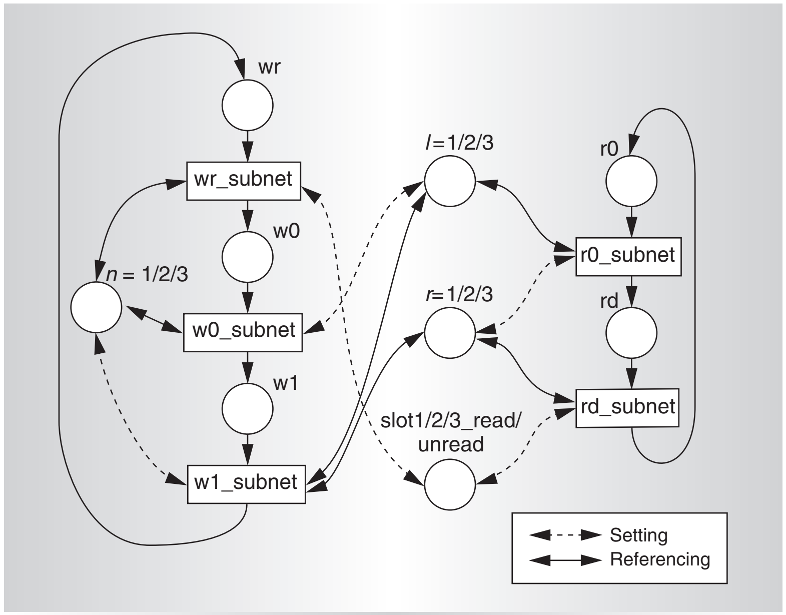


Figure 7. Top-level view of a three-slot Pool's Petri net model.

read access. This specification gives total temporal independence to both reader and writer.

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 restriction implies binary or ternary variables that can be set and reset in hardware within the shortest possible action time. By employing such *control variables*, it's possible to remove any asynchrony from large-scale actions, such as data item access by one of the sides.

In multislot ACM terminology, a *data slot* is a unique portion of shared memory that can contain one data item.[2] It is impossible to implement the Pool ACM specified by Figure 5 with only one data slot and still provide data coherence; simultaneous read and write accesses of multibit data items from the same data slot will likely result in the data's modification after it leaves the writer.

Research has shown that a fully asynchronous Pool of capacity 1 is only implementable using three or more data slots.[2,9,10] In Figure 6, we present a Pool algorithm using three data slots proposed by Simpson.[2]

In ACM implementations that use multiple data slots, the number of data slots should not be confused with ACM capacity. In general, the number of slots should be more than the ACM capacity if any asynchrony between reader and writer is desirable. This requirement does not necessarily present conflicts. For instance, consider a Pool of capacity 1 correctly implemented with three slots. At any time, only one slot contains the nominal current data item in the Pool; the Pool uses the other slots to avoid simultaneous reads and writes of the same physical memory. To correctly implement an ACM with more slots than its capacity requires maintaining data freshness.

In the algorithm in Figure 6, the Pool holds the passing data in one of three data slots labeled slot 1, 2, or 3. Control variables *n*, *l*, and *r* are ternary. This algorithm implies that within the reader and writer processes, the statements must be executed in the order specified, and any statement must not start without the previous one having completed.

Statement w1: $n := \neg(l, r)$ assigns variable n a value different from the current values of l and r. In practice, this differ statement can be implemented by using the matrix $\neg(l, r) = \{(2,3,2), (3,3,1), (2,1,1)\}$. Such a lookup table is easily implementable in hardware.

A theoretical analysis employing Petri net models can establish whether an ACM implementation conforms to the specification. These models highlight important asynchrony and data properties, such as waiting, data coherence, and data freshness. Such modeling and analysis techniques come from previous work.[11]

Algorithms like that in Figure 6 consist of two single-thread cyclic processes, which we can view as finite state machines. Given this view, we can derive Petri net models for such FSMs at the top level, resulting in the Petri net model of the three-slot Pool algorithm shown in Figure 7.

In this model, the subnets are virtual transitions that describe detail actions that result from statements in the writer and reader processes. Shared control variables connect the two FSMs. For any control variable, only one side sets its value, as specified in the algorithm, and the other side reads or references its value. In this high-level net, a single place represents the value of each ternary control variable. This means that the setting and referencing arcs must be bidirectional. The arcs must represent how tokens move back and forth between a statement subnet and the modified/referenced control variable during either a setting or referencing statement. This overview model does not show initial-condition tokens. Earlier work describes this model and the Petri net representations of such properties as data coherence and freshness.[11]

We carried out reachability searches on this model. The results showed that the algorithm maintains data coherence and freshness if and only if statement r0 is atomic with respect to statements w0 and w1. That is, the entire w0 to w1 sequence does not start and finish completely within a single r0 statement and in the order r0 start, w0 and w1 start and finish, then r0 finish. In an implementation, therefore, data coherence and freshness are only maintained under this assumption of relative atomicity.

In practice, this assumption of atomicity is correct in many physical situations. For instance, between statements w0 and w1, the writer might do such obvious tasks as preparing the data item to be written in the next cycle, which should be more time-consuming than setting a control variable. Or an ACM implementation can employ hardware arbitration to ensure relative statement atomicity.[9,10]

A Signal with two data slots

Because a Signal does not provide its reader with full temporal independence, it needs fewer slots to implement than a Pool. We again start from a nonatomic model for the Signal, shown in Figure 8.

This model specifies that the writer can initiate and complete a data access at any time. On the other hand, the reader cannot initiate a data access when a writer access is in progress or when the ACM's data state is inappropriate (when unread = 0). This scheme provides the writer with full temporal independence. Figure 9 shows an algorithm we developed for a two-slot Signal that observes these requirements.[11]

In this algorithm, the reader must wait when $w = r$. The *wait until* clause specifies that

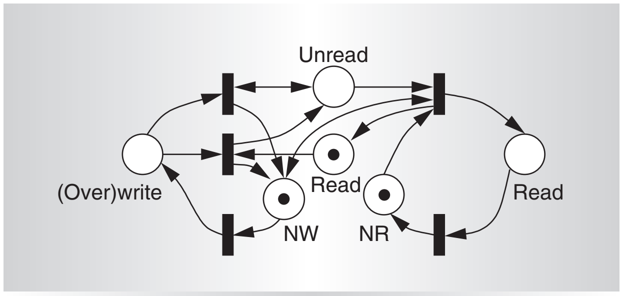


Figure 8. Signal with blocking write and overwrite and non-blocking read.

```
writer                    reader
wr: write slot w          r0: r := r̄
w0: w := r̄                rd: wait until w ≠ r
                              read slot r
```

Figure 9. Two-slot Signal algorithm.

a change in value on *w* should stimulate the reader out of waiting.

We have carried out Petri net modeling and analysis for this algorithm following broadly similar lines as in the case of the three-slot Pool algorithm.[11] Reachability search results confirm that this two-slot Signal algorithm satisfies all requirements, including asynchronism, data coherence, and relaxed data freshness, as long as we can regard control variable statements as atomic.

Hardware designs and simulation

We have implemented both of the preceding ACM algorithms in hardware. The implementations employ speed-independent circuits derived with the direct-translation method based on a type of circuit element known as the David cell.[12]

As discussed earlier, the three-slot Pool algorithm in Figure 6 maintains data coherence and freshness if statement r0 is atomic in relation to statements w0 and w1. These are control variable statements that should take relatively little time to execute compared with the data access statements rd and wr. Thus, a hardware solution that sensibly incorporates this atomicity restriction need not compromise the temporal independence of either writer or reader. Figure 10 (next page) shows the top-level schematic of a hardware design implementing this algorithm.

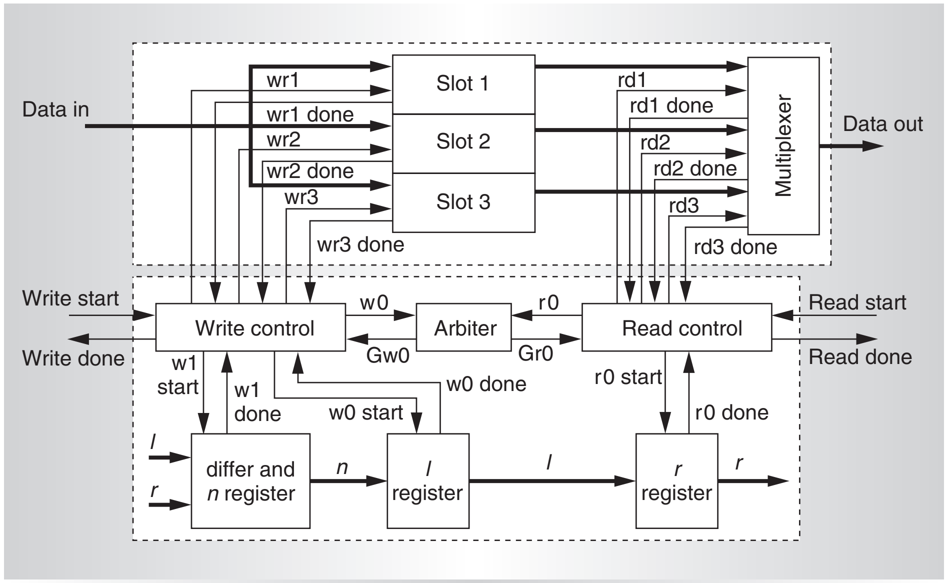


Figure 10. Hardware design implementing a three-slot Pool algorithm.

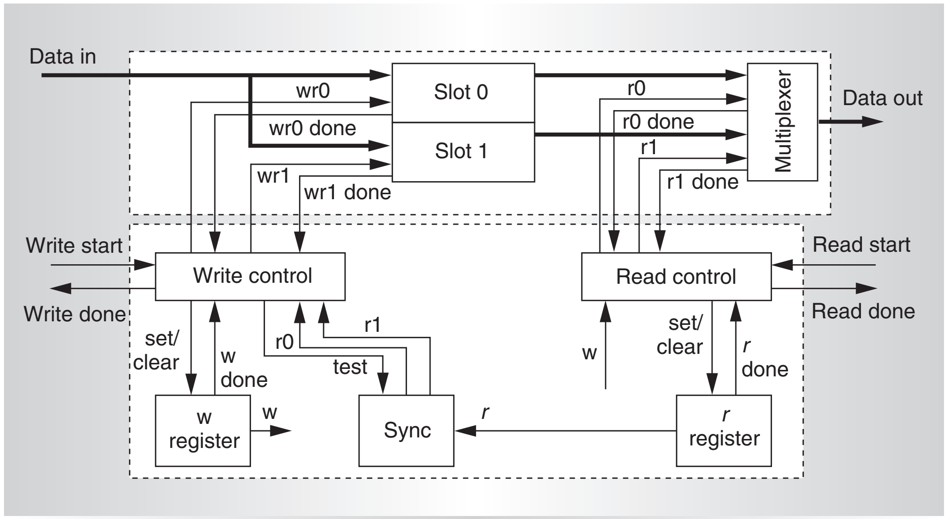


Figure 11. Hardware design of two-slot Signal.

The lower half of Figure 10 shows the control circuits for the three-slot Pool implementation. The figure's top half shows that the circuit uses the control variable values to multiplex and demultiplex the connections to the three slots between the reader and writer. In this way, the access process will connect to the correct slot at each wr and rd statement. The writer and reader control boxes contain logic that manages the sequential statements for the write and read sides. The start/done handshakes, which incorporate completion detection signals fed back from the data slots to the control circuits, ensure that the entire control circuit is speed independent for safety. The registers implementing the control variables are also speed independent and have full completion detection. The arbiter preserves the atomicity of r0 with respect to w0 and w1. We provide a more detailed description of this three-slot Pool hardware implementation in another work.[9]

We have used similar techniques to create a hardware implementation for the two-slot Signal algorithm in Figure 9. Arbitration (via synchronization blocks rather than explicit arbiters) again preserves the atomicity of crucial control variable statements with respect to each other. One point that distinguishes Signals from Pools is the wait-until statement in the part of the algorithm concerned with the reader. Although implementable by polling, the wait-until statement is more energy efficient when implemented by using speed-independent circuits.

Figure 11 shows the hardware block diagram for the two-slot Signal based on Figure 9's algorithm. It shows the data path (in the figure's upper portion) and the control parts (lower portion). We offer a more detailed description of this circuit elsewhere.[10]

Simulation results

The speed independence of circuit designs ensures that the hardware satisfies all the algorithms' requirements, such as maintaining the sequential order of statements within reader and writer processes. To further increase our confidence in the hardware's functionality, we performed both analog and digital simulations on VLSI (0.6-micron technology) imple-

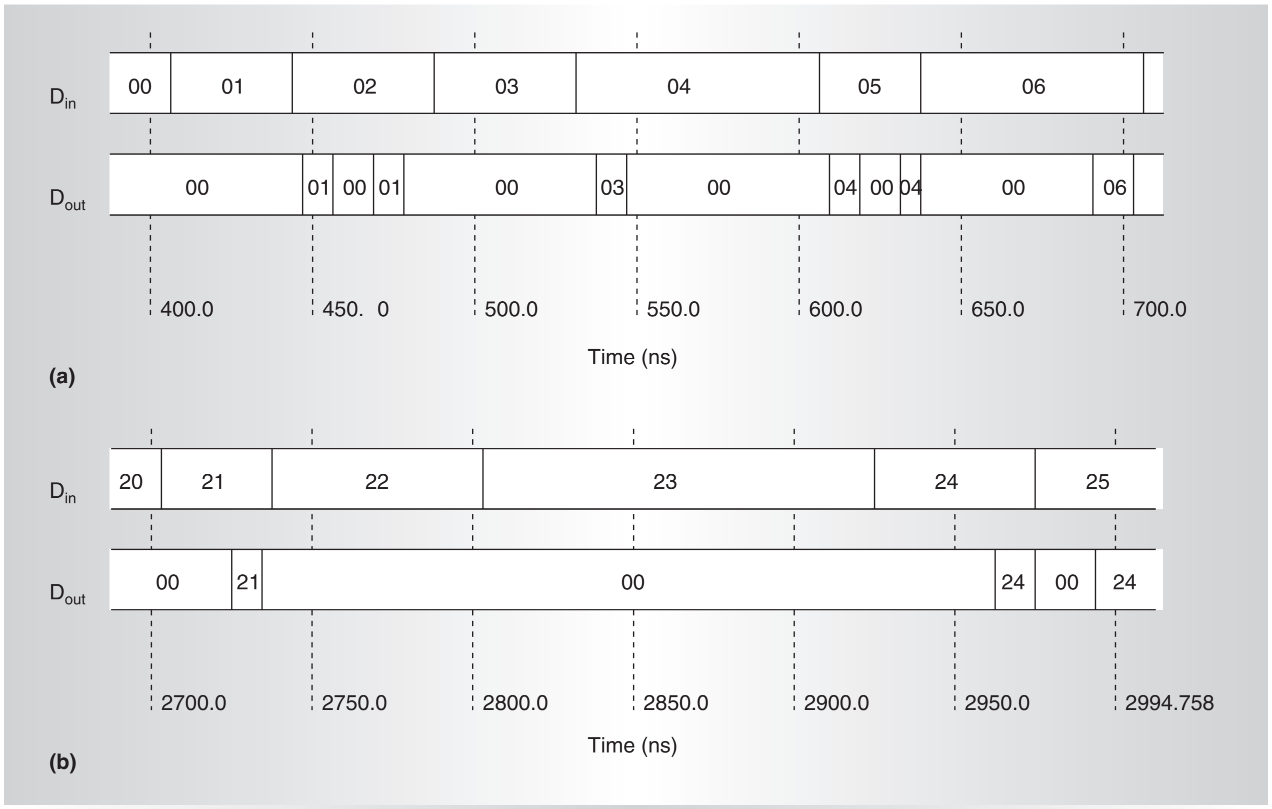


Figure 12. Value sequences of input and output data at the beginning (a) and middle (b) of a simulation.

mentations of the circuit, which we developed using Cadence. We also performed on-chip testing of another Pool ACM circuit, which was fabricated after the Cadence process.[9,10,13] The simulation and on-chip testing corresponded with the theoretical-analysis results and showed that the hardware designs do implement the algorithms.

Figure 12 shows one series of digital simulation results that illustrates the interesting aspect of trading data items for temporal independence in a Pool. These figures show the values of data items being written into and extracted from the Pool by the writer and reader. The 00 parts are spacer states on the reader's side; these spacers denote times when the reader is not carrying out a data access. Random-number generators are used for dynamically varying the duration of reader and writer cycles. The ACM can maintain both data freshness and data coherence even under such variable operating modes. These sequences clearly show overwriting and rereading; other recorded data from individual signal wires also showed that neither side suffers an ACM-imposed delay. These results are consistent with the specification of a Pool providing full temporal independence for both reader and writer.

Other related work strengthens the foundation for the het-based methodology of real-time and embedded systems with potential SOC applications. This work includes quantitative analyses of ACMs.[14] Rather than rely on simulations, these analyses showed it's possible to analytically derive numerical results for the rate of data loss and rereading with the help of Petri net models.

Het-based systems provide for flexible interfaces between elements with self-motivated and reactive timing. This flexibility could facilitate large-scale integration without resorting to higher-level manipulations of time through techniques such as those found in traditional networks (Ethernet, ATM, bus management).

In the future, we plan to extend the work we present here by further studying the effects of ACMs in systems at higher levels and investigating the behavior of systems employing ACMs. It is important to establish a theoretical framework for systems containing ACMs at higher levels before such techniques can be applied with confidence. MICRO

Acknowledgments

We carried out this work with the support of the Engineering and Physical Sciences Research Council (EPSRC), UK (project Coherent—http://async.org.uk/coherent/—at Newcastle University, GR/R32666, and at Kingston University, GR/R32895, and project Besst (http://async.org.uk/besst/) at Newcastle University, GR/R16754. We thank Alex Bystrov, Eric Campbell, Tony Davies, Hugo Simpson, and Walter Vogler for their valuable discussions.

References

1. L. Lamport, "On Interprocess Communication: Parts I and II," *Distributed Computing*, vol. 1, no. 2, 1986, pp. 77-101.
2. H.R. Simpson, "Four-Slot Fully Asynchronous Communication Mechanism," *IEE Proc.,* vol. 137, no. 1, Jan. 1990, pp. 17-30.
3. H.R. Simpson and E. Campbell, "Real-Time Network Architecture: Principles and Practice," *Proc. Asynchronous Interfaces: Tools, Techniques and Implementations* (AINT 00), Technische Universiteit Delft, the Netherlands, 2000, p. 5 and handouts.
4. D.M. Chapiro, *Globally-Asynchronous Locally-Synchronous Systems,* doctoral dissertation, Dept. of Computer Science, Stanford Univ., Palo Alto, Calif., 1986.
5. K.Y. Yun and A.E. Dooply, "Pausible Clocking Based Heterogeneous Systems," *IEEE Trans. Very Large-Scale Integration (VLSI) Systems*, vol. 7, no. 4, Dec. 1999, pp. 482-488.
6. T. Chelcea and S.M. Nowick, "Robust Interfaces for Mixed-Timing Systems with Application to Latency-Insensitive Protocols," *Proc. 38th Design Automation Conf.* (DAC 01), ACM Press, New York, 2001, pp. 21-26.
7. L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli, "The Theory of Latency Insensitive Design," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* IEEE Press, Piscataway, N.J., 2001, pp. 1059-1076.
8. J.L. Peterson, *Petri Net Theory and the Modelling of Systems*, Prentice-Hall, Upper Saddle River, N.J., 1981.
9. F. Xia et al., "Asynchronous Communication Mechanisms Using Self-Timed Circuits," *Proc. 6th Int'l Symp. Advanced Research Asynchronous Circuits and Systems* (ASYNC 00), IEEE CS Press, Los Alamitos, Calif., 2000, pp. 150-159.
10. A. Yakovlev, F. Xia, and D. Shang, "Synthesis and Implementation of a Signal-Type Asynchronous Data Communication Mechanism," *Proc. 7th Int'l Symp. Asynchronous Circuits and Systems* (ASYNC 01), IEEE CS Press, Los Alamitos, Calif., 2001, pp. 127-136.
11. F. Xia and I. Clark, "Algorithms for Signal and Message Asynchronous Communication Mechanisms and Their Analysis," *Proc. 2nd Int'l Conf. Applications of Concurrency to System Design* (ACSD 01), IEEE CS Press, Los Alamitos, Calif., 2001, pp. 65-74.
12. D. Shang, F. Xia, and A. Yakovlev, "Asynchronous Circuit Synthesis via Direct Translation," *Proc. Int'l Symp. Circuits and Systems* (ISCAS 02), IEEE Press, Piscataway, N.J., 2002, pp. 369-372.
13. D. Shang, F. Xia, and A. Yakovlev, "Testing a Self-Timed Asynchronous Communication Mechanism (ACM) VLSI Chip," *IEEE Workshop Design and Diagnostics of Electronic Circuits and Systems* (DDECS 01), IEEE Press, Piscataway, N.J., 2001, pp. 53-56.
14. A. Madalinski, F. Xia, and A. Yakovlev, "Studying the Data Loss and Data Re-Reading Behaviour of a Four-Slot ACM Using SPN Techniques," *Proc. 7th UK Async Forum*, 1999; http://www.cs.man.ac.uk/async/events/ukforum.html.

Fei Xia is a research associate with the School of Electrical, Electronic, and Computer Engineering at the University of Newcastle upon Tyne, UK. His research interests include the design, modeling, and analysis of real-time asynchronous systems and in particular asynchronous data communication in such systems. Xia has a BEng in automation from Tsinghua University, an MSc in control systems from the University of Alberta, and a PhD in electronic engineering from King's College, University of London.

Alex V. Yakovlev is a professor with the School of Electrical, Electronic, and Computer Engineering at the University of Newcastle upon Tyne. His research interests include the modeling and design of asynchronous, concurrent, real-time, and dependable systems. Yakovlev has an MSc and PhD in computer science from St. Petersburg Electrotechnical University. He is a member of the IEEE.

Ian G. Clark is a research associate with the School of Electrical, Electronic, and Computer Engineering at the University of Newcastle upon Tyne. He is also a senior researcher at Kingston University, UK. His research interests include the design, modeling, and analysis of real-time asynchronous systems, GALS, and asynchronous communication mechanisms. Clark has a BSc (Eng) and a PhD in electronic engineering from King's College, University of London. He is a member of the IEE.

Delong Shang is a research associate with the School of Electrical, Electronic, and Computer Engineering at the University of Newcastle upon Tyne. His research interests include computer system and VLSI circuit design and testing, especially in asynchronous systems. Shang has a BSc in computer science and technology from Nanjing University and an MEng in computer engineering from the Chinese Academy of Sciences.

Direct questions and comments to Fei Xia at the School of Electrical, Electronic, and Computer Engineering, University of Newcastle upon Tyne, NE1 7RU, UK; fei.xia@ncl.ac.uk.

For further information on this or any other computing topic, visit our Digital Library at http://computer.org/publications/dlib.