A COMPARISON OF SOME WAIT-FREE COMMUNICATIONS MECHANISMS

Ian G. Clark, Anthony C. Davies
Department of Electronic Engineering,
King's College London, Strand, London, WC2R 2LS, England
ian.clark@kcl.ac.uk, tonydavies@kcl.ac.uk

Abstract

This paper presents an overview and assessment of those fully-asynchronous interprocess communications mechanisms (ACMs) which can operate in a wait-free manner. Using such ACMs can avoid temporal constraints between processes (of the kind needed to implement a synchronised rendezvous) which may lead to deadlock and unacceptable performance. The method of analysis and the comparisons use reachability-searches carried out on Petri Net models of each mechanism, and this enables the correctness to be demonstrated and/or conditions leading to failure to be identified, and so provides a useful tool to aid the designer. ACMs may become important because increased use of asynchronous designs is likely in future complex integrated circuit chips and mobile multimedia systems.

1. Introduction

Asynchronous designs are likely to be adopted to a greater extent in future, as the speed and complexity of integrated circuits increases. Compared to synchronous designs, they offer the advantages of lower power consumption (since switching transients are needed only when computation needs to be performed) and better electromagnetic compatibility (because of the elimination of a high-power, high-frequency clock). In any case the difficulties of distribution of a high-frequency clock across a complex integrated circuit may make fully synchronous designs impracticable. Such integrated circuit designs are likely to comprise a number of locally synchronous processors communicating asynchronously.

Mobile multimedia developments imply a distributed multiprocessor system, with some components in relative motion, which also makes fully synchronous designs inappropriate.

The question of reliable interprocess communications between independent and unsynchronised processors is therefore one which needs to be addressed. It is often assumed that such interprocess communications necessarily involve some form of synchronisation (for example, the ADA rendezvous) which implies waiting, and a risk of deadlock in the case of either faulty design or component failure. Associated with such process synchronisation is the need for mutual exclusion and hence arbitration in the access to shared resources. The possibility of designing mechanisms to implement wait-free communication between processes (whether on the same or different processors) is often assumed impossible, and the need for arbiters in the access to the associated shared memory is assumed inevitable.

However, for one particular class of interprocess communications (corresponding to a MASCOT 'pool' [1] for which the write operation is destructive and the read operation is non-destructive), a number of wait-free schemes have been proposed [2 - 12], in most cases independently of one another, with the same objective, but based upon widely differing concepts and analysed by their authors using radically different methods. Some of the proposals have overlooked important properties that such mechanisms ought to satisfy. For example, Kirousis *et al* [9] make no reference to any kind of freshness property. The proposed designs are typically rather complicated and difficult to understand, and it is by no means easy to be confident about their correctness.

In this paper, the form of interprocess communication which can be wait-free is explained, necessary properties of such mechanisms are reviewed and some of the wait-free asynchronous communications mechanisms (ACMs) are compared. A uniform method of analysis is used, based upon Petri Nets and involving a state-reachability search, which enables any of the schemes for ACMs to be checked for various correctness properties and which is capable of revealing the failure modes where these exist.

2. Wait-free asynchronous communications

For the producer-consumer type of communication (corresponding to the MASCOT 'channel') it is not possible to have a wait-free mechanism. The reader (consumer) must wait if the writer (producer) has not yet produced new data, and the writer may have to wait if the storage area (buffer, queue) between the processes is full.

However, when accessing 'reference data' (corresponding to the MASCOT 'pool'), there is no requirement for the reader to read every item written. Missing out data items is acceptable as long as the

'newest' data is available to the reader. There may be many writes between two reads or many reads between two writes. The writer does not have to wait for the reader to 'consume' any data before writing new data and therefore may be given the freedom to write new data at any time. The implementation problem arises if a read and a write can partially or completely overlap in time (as they may in any fully asynchronous multi-processor system), since it is not possible to simultaneously read from and write to the same memory location - at least not without data corruption.

A mechanism to implement such communications must satisfy some simple behavioural properties:

1. *Coherence*

Each data item being communicated is typically a multi-bit digital word or a record with several fields. It is important that the reader always obtains a coherent item, as opposed to one which consists partly of an 'old' item and partly of a 'newer' item. If the data is a record with multiple fields, either none or all must be updated in any item obtained by the reader.

For example, if a sequence of time values were being communicated, such as ...07:52, 07:58, 08:01, 08:06, 08:12 ... then reading values such as 08:58 and 07:06 would correspond to incoherent data, and could be more seriously incorrect than getting 'late' data (such as 07:58 when the time was really 08:01)

2. *Sequencing*

It is assumed that once a given data item has been read, if a subsequent read were to acquire an earlier item, the latter would effectively be 'out of date' and therefore incorrect. The data items should therefore be read in the order written, even though it is not necessary to read every item and permissible to read the same item more than once. Correct sequencing requires that '..new, old, new..' is not possible, and once an item has been read, subsequent reads must never access an earlier item.

Thus, taking the same example of time values, reading ...08:01, 07:58: 08:01, 08:12, 08:12... would correspond to out of sequence data.

3. *Freshness*

A mechanism could comply with the first two properties by repeatedly providing the first item written (07:52) to the reader, and never transmitting any subsequent value. Clearly that is unacceptable, so the properties of data coherence and sequencing are necessary but not sufficient.

The data items must not be 'out of date'. This is a rather more complex property to define precisely. When the writer has completed the writing of a new data item, it may not immediately be available to the reader, because internal control variables denoting its availability and memory location may still be in the process of being updated when the reader attempts to read. It would then be correct for the reader to obtain the immediately previous data item, which would be defined as the 'freshest' item though an 'even more fresh' item was stored in the mechanism.

These definitions have been discussed and defined carefully by Simpson [5,6,7]. Any mechanism design which fails to satisfy these three properties cannot be regarded as correct.

Of course, there are specific applications where failure to comply with one or more of these properties may be acceptable, but for an ACM for universal use they are necessary and appear to be sufficient.

In addition, in order to comply with these three behavioural properties, the mechanism should satisfy some operational properties. For example in the transfer of data between asynchronous processes, there is an inevitable risk that occasionally the timing relationships between the processes will be such that set up or hold times of latches are not complied with, and this or some other timing condition results in a metastable transient arising in the mechanism control variables. Although the risk of metastablity in an asynchronous transfer cannot be ruled out, there are design choices for the mechanism which minimise its effects (typically by exchanging value uncertainty with delay in making a decision about the value). If a metastable signal level is associated with a junction where fan-out occurs, the metastable output value may be interpreted as a logic *zero* by some subsequent inputs and as a logic *one* by others. Thus, fan-out in the control-variable path is best avoided. By ensuring that control variables are all single-bits, metastablity can at worst change a *zero* to a *one* or vice versa, whereas with multi-bit variables, a metastability error on one bit can have a more serious or unpredictable effect. The timing relationships are such that metastability in the data transfer path is most unlikely to occur because of the control-variable steering of the ACM.

If metastability is ignored, many proposed ACM's appear correct. However, when metastablity is allowed for, a more careful analysis is needed and often reveals unsatisfactory modes of behaviour apparently overlooked by some ACM designers.

Of course, constructional properties must also be considered - such as complexity, ease of implementation, suitability for software or hardware implementation, and avoidance of complicated instructions or data-structures.

3. The number of slots required for fully asynchronous communication

It is immediately obvious that at least two data areas ('slots' or 'tracks') are required, one to be used by the reader while the other is used by the writer. A shared control variable is needed to inform the reader

to which slot new data has been written. Because of the need for time to update control variables, two slots are not sufficient, and at least one more is required.

A basic three-slot algorithm is given in Appendix A. When given two ternary values as operands, the function 'differ' returns a value not equal to either of the supplied operands. As shown in Table I, this algorithm does not even maintain the data coherence property (for which reason the other properties were not checked).

All known three-slot algorithms fail for some timing relationships, but there are a number of correct implementations using four slots. Some three-slot algorithms have been designed which work correctly within well-defined timing-constraints, but of course these do not qualify as fully-asynchronous [16].

4. Modelling and analysis of the algorithms

The originators of the algorithms have each used very different methods to describe their operation and to explain their designs. In addition, methods used to justify correctness are very varied, and sometimes difficult to understand without very detailed study.

We use a Petri Net model for each algorithm, to which extra states are added corresponding to each behavioural property being examined (failure to preserve coherence, sequencing, and freshness) or to assess the possible result of metastability [13].

An exhaustive reachability search is then performed, and if any state corresponding to faulty behaviour is reachable (from any initial state) then this shows that the mechanism being investigated has failed to comply with the required property. Moreover, the results enable the particular timing-sequence(s) or interleavings which result in the faulty behaviour to be identified. Because the reader and writer processes are cyclic and not intended to terminate, a modified search method had to be developed in order to prevent the creation of infinite search-trees.

A breadth-first search method is used which finds the shortest route to any 'bad markings' (states which indicate a failure of the mechanism). A list is maintained of all states visited, and each state encountered is checked against the list to ensure that it is a new state before proceeding along the search-route.

The number of states its typically too large to search for all possible faults using a single model. Therefore, a different model is used for each property, e.g. one model to search for coherence faults, another model to search for sequencing faults, and so on. Setting up the models is straightforward and newly devised mechanisms or variations in existing designs can easily be assessed. In particular, this enables the effect of design changes to be quickly investigated, and the effect of restrictions on timing relationships between reader and writer to be assessed.

5. Classification of the Algorithms

Typically the algorithms consist of several data slots and some control variables (Fig. 1). Some of the control variables must be shared between the writing and reading processes, and others may be local to the processes within which they are defined. Moreover, some mechanisms have read-only access to certain of the shared variables from one side or the other.

A simple measure of storage size is to simply count the number of bits required, and express it in the format $x.b+y$ where x is the number of slots, b is the number of bits in each data item, and y is the number of control bits.

The complexity of the instructions forming the algorithm may be compared - if there are conditional branches or iterative loops within the algorithm, or complex operations on the control-variables (e.g. other than simple assignments or logical operations) this may lead to implementation difficulties and, more importantly, behaviour in the presence of metastability which is difficult to analyse or predict.

Example results

Of the many published algorithms, six very different ones have been chosen to illustrate the results obtained. Table I shows a comparison of the model-complexity involved, and also illustrates that one of the algorithms fails to satisfy the data sequencing and freshness properties and another fails to satisfy the coherence property. Not maintaining coherence seems to be such a serious failing that the assessment of the other two properties was not carried out. The Table entries represent the complexity of verifying their correctness of each ACM, but do not represent the complexity of implementation of the mechanisms. However, they do give a rough indication of implementation-complexity. Table II shows how the control variables are allocated.

The algorithms are presented by their inventors in very different formats and styles and are explained in very different ways. To fully understand the way that they were developed and the reasoning behind their design and operation it is therefore necessary to study these original descriptions. In most cases, some common features exist, in the form of control variables which steer the writer away from any slot currently being accessed by the reader, and which ensure that the writer does not repeatedly write to the same slot. The control variables also have to provide a mechanism for the writer to signal to the reader when data has been written to a slot and to direct the reader to this slot for the next read operation.

One of Simpson's algorithms is shown in Appendix B. Note that the variable called 'slot' is not the data-storage slot but is used to hold slot identification numbers. The storage size is 4*b*+7. For details of this algorithm and the reasoning which led to this form, see Simpson [5,6].

If implemented in hardware, simplification is possible, and some writer instructions can be executed in parallel. It has been implemented in integrated circuit form by Campbell [14, 15]

Appendix C shows a version of an algorithm proposed by Tromp [4] which has a storage size 4*b*+14. The structure can be seen to be rather more complicated than the Simpson algorithm, with no apparent compensating advantage.

Kirousis *et al* [9] proposed a number of ACM designs, of which the best appears to be one with storage size 4*b*+39. These control variables are multi-valued so the number of bits exceeds the number of variables. As can be seen from Appendix D, it is substantially more complex in structure than the other two, and includes some record-like data structures in the control variables (arrays in which each element is a tag/track pair). The algorithm also includes several conditional statements and some complex operations such as modulo-3 arithmetic and selecting a minimum of several values. Moreover, its authors appear to have overlooked the need for the data freshness property and our analysis of their mechanism indicates that it fails to maintain freshness. To understand the notation and operations involved it is necessary to refer to the paper of Kirousis *et al* [9].

Appendix E shows an algorithm proposed by Anderson *et al* [10]. The original version uses a two-bit coding of data which is needed for some other applications but which is unncessary when used only as an ACM. Therefore we have made some simplifications, resulting in a storage size of 4*b*+9.

Finally, Appendix F shows an algorithm due to Peterson *et al* [5]. The tests for the three behavioural properties had to be altered because the original version writes to multiple slots, with no automatic steering logic as provided by all the other ACMs. The algorithm tries to detect whenever an attempt is made to write to and read from the same slot simultaneously, and then uses a different copy of the data-item.

The algorithms are all presented in a uniform high-level style in these Appendixes. This is intended to make them easier to compare. Presentation in this form involved making some changes to the originally-published formats. For an actual implementation, whether in hardware or software, a number of further changes and optimisations are possible, in order to improve the implementation efficiency.

Conclusions

Although many independent proposals for fully asynchronous communications mechanisms have been published, they are difficult to compare and assess on the basis of these published designs and analyses. The Petri Net based reachability search technique provides a way of comparing them and reveals many features of their behaviour, some of which are not obvious or otherwise easy to discover. The method provides a useful tool to aid the designer of ACMs.

Acknowledgements

The U.K. Engineering and Physical Sciences Research Council (EPSRC) is thanked for financial support (Grant No. GR/L92471 †), and colleagues at King's College London, the University of Newcastle-upon-Tyne, and BAE Systems are thanked for interesting discussions about the topic

† see `http://www.eee.kcl.ac.uk/~comfort`

References

[1] JIMCOM et al. 'The Official Handbook of MASCOT', Version 3.1, June 1987

[2] Lamport L. 'On Interprocess Communication. I. Basic Formalism and II. Algorithms', *Distributed Computing*, 1986, 1, 77-101

[3] Li., M., Tromp J. and Vitanyi P.M.B 'How to Construct Wait Free Variables', *Proc. International Colloquium on Automata, Languages, and Programming*, in *Lecture Notes in Computer Science*, 1989, 372, 488-505

[4] Tromp J. 'How to Construct an Atomic Variable (Extended Abstract)', *Proc. Int. Workshop on Distributed Algorithms*, 392, 1989, 292-302

[5] Peterson G.L. 'Concurrent reading while writing', *ACM Transactions on Programming Langauges and Systems*, 5, 1983, 46-55

[6] Simpson H.R. 'Four-slot fully asynchronous communication mechanism', *IEE Proceedings, Part E, Computers and Digital Techniques*, 1990, 37, 17-30

[7] Simpson H.R. 'Correctness analysis for class of asynchronous communication mechanisms', *IEE Proceedings, Part E, Computers and Digital Techniques*, 1992, 139, 35-49

[8] Simpson H.R. 'New algorithms for asynchronous communication', *IEE Proceedings, Part E, Computers and Digital Techniques*, 1997, 144, 227-231

[9] Kirousis L.M., Kranakis E. and Vitanyi P.M.B. 'Atomic Multireader Register', *Proc. International Workshop on Distributed Computing*, 1988, 278-296

[10] Anderson J.H. and Gouda M.G. 'A Criterion for Atomicity', *Formal Aspects of Computing*, 1992, 273-298

[11] Haldar S. and Subramanian P.S. 'Space optimum conflict-free construction of 1-writer 1-reader multivalued atomic variable', *Proc. International workshop on Distributed Algorithms*, 857, 1994, 116-129

[12] Singh A.K., Anderson J.H. and Gouda M.G. 'The Elusive Atomic Register', *Journal of the ACM*, 41, 1994, 311-339

[13] Clark I.G., Xia F., Yakovlev A.V. and Davies A.C. 'Petri net models of latch metastability', *IEE Electronics Letters*, 1998, 34, 635-636

[14] Campbell E. 'DIADEM hardware description', British Aerospace Defence Dynamics Ltd. Report, 1991

[15] Campbell E. 'DIA temporal characteristics and their experimental verification', British Aerospace Defence Dynamics Ltd., Univ. of York, Admiral Management Services, 1992

[16] Xia F., Yakovlev A.V, Shang D., Bystrov A., Koelmans A. and Kinniment, D.J. 'Asynchronous comunications mechanisms using self-timed circuits', *Proc. 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async 2000)*, April 2000, Eilat, Israel.

Table I Comparison of Petri Net models and analysis of four ACM algorithms

	Coherence				Sequencing				Freshness			
	T	**P**	**S**	**B**	**T**	**P**	**S**	**B**	**T**	**P**	**S**	**B**
basic 3-slot	33	31	324	6	-	-	-	-	-	-	-	-
Simpson	54	39	1472	0	214	134	13586	0	124	80	9076	0
Tromp	96	53	13712	0	329	173	146751	0	212	119	303748	0
Anderson	40	37	1517	0	200	132	16561	0	116	95	31885	0
Peterson	53	53	54460	1280	-	-	-	-	-	-	-	-
Kirousis	293	149	1201211	0	441	234	1993295	105	336	176	1679898	35

T = transitions **P** = places **S** total states **B** bad states

Table II Comparison of control variables in four ACM algorithms

	shared variables			W only	R only	type
	r/o to W	r/o to R	r/w to W and R			
basic 3-slot	1	1	0	1	1	ternary
Simpson	1	3	0	2	1	bit
Tromp	2	6	0	3	3	bit
Anderson	1	3	0	3	2	bit
Peterson	1	3	0	0	5	bit
Kirousis	3	3	0	6	4	multi-valued

r/o = read-only r/w = read/write

Appendix A Basic Three-slot (3*b*+ 8) algorithm

```
mechanism basic_three_slot is
     T : array[1..3] of data := (null, null null);
     latest, reading : 1..3 := 1;

  procedure write(item : in data) is
    index : 1..3;
  begin
    index := differ(latest, reading);
    T[index] := item;
    latest := index;
  end write;

  function read return data is
     index : 1..3;
  begin
    index := latest;
    reading := index;
    return T[index];
  end read;
end basic_three_slot;
```

Appendix B The Simpson (4*b*+ 7) algorithm

```
mechanism simpson_four_slot is
     T : array[bit, bit] of data := ((null, null),(null,null));
     slot : array[bit] of bit := (0,0);
     latest, reading : bit := 1,1;

  procedure write(item : in data) is
    pair, index : bit;
  begin
    pair := not reading;
    index := not slot[pair];
    T[pair, index] := item;
    slot[pair] := index;
    latest := pair;
  end write;

  function read return data is
    index : bit;
  begin
    reading := latest;
    index := slot[reading];
    return T[reading, index];
  end read;
end simpson_four_slot;
```

Appendix C The Tromp (4*b*+14) algorithm

```
mechanism tromp_four_slot is
     T   : array[bit, bit] of data;
     D   : array[bit, bit] of bit;
     R, W: array[bit] of bit;

 procedure write(item : in data) is
     x : array[bit] of bit;
     w : bit;
  begin
    if R[not w] = W[not w] then
          w := not w;
          T[w, x[w]] := item;
          W[not w] := not W[not w];
    else
          x[w] := not x[w];
          T[w, x[w]] := item;
          D[w, not x[w]] := not D[w, not x[w]}
    endif;
  end write;

 function read return data is
     d : array[bit] of bit;
     r : bit;
 begin
   if W[r] <> R[r] then
        R[r] := not R[r];
        r := not r;
        d[0] := D[r,0];
        d[1] := D[r,1];
     else
         d[d[0] xor d[1]] := D[r, d[0] xor d[1]];
     endif;
     return T[r, d[0] xor d[1]];
 end read;
end tromp_four_slot;
```

Appendix D TheKirousis (4*b*+39) algorithm

```
mechanism kirousis_four_track is
  TRACK : array[1..4] of data := (null, null, null, null);
     RQ : array[0..2] of {P, E} := (E, E, E);
      A : array[0..2] of (E),(0,1),(0,2),(0,3),(0,4),(1,1),(1,2),(1,3),(1,4),
                         (2,1),(2,2),(2,3),(2,4)} := (E, E, E);

     procedure write (item : in data) is;
       wl, wtg : {0,1,2} := 0,0;
       wt, wt_ : {1,2,3,4};
            vb : array[bit] of {E,1,2,3,4} := (E, E);
            rq : array[0..2] of {P, E}
     begin
         rq_wl := RQ_wl;
         if rq_wl = P then
             wl := wl + 1;
             vb := {wt, wt_};
         endif;
         wt_ := wt;
         wt := min({1,2,3,4} - ({wt_}∪ vb));
         TRACK[wt] := item;
         wtg := wtg + 1;
         A_wl+1 := E;
         A_wl := (wtg, wt);
     end write;

     function read return data is;
       s   : (E),(0,1),(0,2),(0,3),(0,4),(1,1),(1,2),(1,3),(1,4),
             (2,1),(2,2),(2,3),(2,4)};
       rt  : {1,2,3,4};
       rl  : {-1,0,1,2} := -1;
       rtg : {0,1,2} := 0;
     begin
        rl := rl + 1;
        a_rl := A_rl;
        if a_rl = E then
           rl := rl - 1;
           a_rl := A_rl;
           if(a_rl <> E) and (a^tag_rl = rtg + 1) then
              (rtg, rt) := a_rl;
           endif;
        else
           s := a_rl;
           RQ_rl+1 := E;
           RQ_rl := P ;
           a_rl := A_rl;
           if a_rl <> E then
             (rtg, rt) := a_rl;
           else
             (rtg, rt) := s;
           endif;
        endif;
        return TRACK[rt];
     end read;
end kirousis_four_track;
```

Appendix E The Anderson (4*b*+9) algorithm

```
mechanism anderson_four_slot is
     Y   : array[bit, bit] of data := ((null, null), (null, null));
     Z   : array[bit] of bit := (0, 0);
     WP, RP : bit := 0, 0;

  procedure write(item : in data) is
      d, wp, walt : bit := 0, 0, 0;
  begin
    d := RP;
    wp := not d;
    walt := not walt;
    Y[wp, walt] := item;
    Z[wp] := walt;
    WP := wp;
  end write;

  function read return data is
      ralt, rp : bit := 0, 0;
  begin
    rp := WP;
    RP := rp;
    ralt := Z[rp];
    return Y[rp, ralt];
  end read;
end anderson_four_slot;
```

Appendix F The Peterson (5*b*+9) algorithm

```
mechanism peterson_five_slot is
     buff1, buff2, copybuff : data := null, null null;
     switch, wflag, writing, reading : bit := 0,0,0,0;

  procedure write(item : in data) is
  begin
     wflag := 1;
     buff1 := item;
     switch := not switch;
     wflag := 0;
     if reading = writing then
          copybuff := item;
          writing := reading;
     endif;
     buff2 := item;
  end write;

  function read return data is
      buff1_copy, buff2_copy : data;
      bad1, sswitch, sswitch2, sflag, sflag2: bit := 0, 0, 0, 0, 0
  begin
    reading := not writing;
    sflag := wflag;
    sswitch := switch;
    buff1_copy := buff1;
    sflag2 := wflag;
    sswitch2 := switch;
    buff2_copy := buff2;
    bad1 := sswitch xor (sswitch2 or sflag or sflag2);
    if reading = writing then return copybuff;
         elseif bad1 then return buff2_copy;
         else return buff1_copy;
    endif;
  end read;
end peterson_five_slot;
```
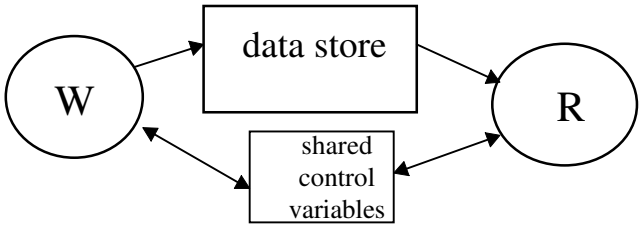
Figure 1. Configuration of ACM