The Design of a Self-timed Circuit for Distributed Mutual Exclusion

Alain J Martin

Computer Science
California Institute of Technology

5097:TR:83

THE DESIGN OF A SELF-TIMED CIRCUIT for DISTRIBUTED MUTUAL EXCLUSION

Alain J. Martin

Computer Science Department
California Institute of Technology

5097:TR:83

The research described in this paper was sponsored by
the Defense Advanced Research Projects Agency, ARPA Order No. 3771,
and monitored by the Office of Naval Research
under contract number N00014-79-C-0597

© California Institute of Technology, 1983

to appear in
Proceedings, 1985 Chapel Hill Conference on VLSI

The Design of a Self-timed Circuit for Distributed Mutual Exclusion

Alain J. Martin
Department of Computer Science
California Institute of Technology
Pasadena CA 91125

1. Introduction

The purpose of this paper is twofold: to describe a design method for self-timed systems, and, as an illustration of the use of the method, to derive a self-timed circuit for distributed mutual exclusion.

The method consists in "compiling" a high-level description of the solution—i.e. a set of communicating parallel processes—into a circuit—i.e. a network of elementary operators (*and*, *or*, C-element, arbiter, and flip-flop). The compilation is systematic and essentially relies upon the four-phase handshaking expansions of the communication actions. The program of each process is compiled into a set of production rules from which all sequencing has been removed. By matching these "production rules" to those describing the semantics of each operator, the programs are identified with networks of operators.

In order to convince the reader that the method presented is applicable to other than trivial examples, we have chosen to illustrate it with a difficult and relevant problem. We believe that the circuit derived by applying the method presents all aspects of a "good" (and new!) solution. The specification of the circuit is the following: An arbitrary number (> 1) of cyclic automata, called "masters", make independent requests for exclusive access to a shared resource. The circuit should handle the requests from the masters in such a way that

1) any request is eventually granted
2) there is at most one master using the shared resource at any time.

The masters are independent of each other: they don't communicate with each other, and the activity of a master not using the resource should

not influence the activity of other masters.

The circuit should be as *self-timed* or *speed-independent* (C. L. Seitz [6]) as possible. By definition, a self-timed circuit is partitioned into *isochronic* (or *equipotential*) regions: Inside an isochronic region, communication along a wire is instantaneous; communication between isochronic regions can take an arbitrary amount of time. The smaller the size of the isochronic regions relative to the size of the circuit, the more speed-independent the circuit. The circuit should be distributed: it should be a collection of "servers"—one per master—communicating with each other by handshaking (no shared data, and of course no shared clock). Further, no activity should be going on in the circuit when there is no request for the shared resource.

The relevance of the mutual exclusion problem in systems exhibiting concurrency need not be advocated any longer. As we shall see, arbitration among asynchronous signals (i.e. choice of one out of several) is an important part of the mutual exclusion algorithm. Since the metastable properties of arbiters ([1], [6]) make it impossible to put a limit on the amount of time an arbiter takes to reach a stable state, the self-timed character of the solution is a great advantage. Moreover, both the self-timed and distributed properties of the solution facilitate its inclusion into distributed systems.

2. The distributed mutual exclusion algorithm

A master M communicates with its private server m. When M wants to use the shared resource —M is said to be *candidate*— it issues a request to m. When the request is accepted, M uses the resource (for a finite period of time), then informs m that the resource is free again.

The servers are connected in a ring. At any time exactly one (arbitrary) server holds a "privilege". Only the "privileged server" may grant the resource to its master thereby guaranteeing mutual exclusion on the access to the resource. A non-privileged server transmits a request from its master—or from its left-hand neighbor—to its right-hand neighbor. A request circulates to the right (clockwise) until either it reaches a server whose master is candidate (this server ignores the request until it has served its master) or it reaches the privileged server. The privileged server reflects the privilege to the left (counter-clockwise) until it reaches the server that generated the request. This server then becomes privileged, and may grant the resource to its master. The strategy consisting of passing requests clockwise and reflecting the privilege counterclockwise has two important advantages. First, no Boolean message need actually be transmitted. Second, no message need be reflected: the completion of a pending request is interpreted as passing the privilege.

The "high-level" description of the solution is a set of communicating concurrent processes. This is the reference description: it is this description that must be proved correct. All other descriptions that will be generated at the different steps of the "compiling" procedure are automatically correct since each is derived from the previous one by semantics-preserving transformations.

2.1 The programming notation

For the sequential part of the algorithm, we use a subset of E. W. Dijkstra's guarded command language [2], with a simpler syntax. Since it is not the purpose of this paper to deal with formal semantics and correctness proofs, we give only a very informal definition of the semantics of the constructs used.

i) $b\uparrow$ stands for $b :=$ **true**, $b\downarrow$ stands for $b :=$ **false**.

ii) The execution of the statement $[G1 \rightarrow S1 \mid G2 \rightarrow S2]$, where $G1$ and $G2$ are Boolean expressions, and $S1$ and $S2$ are arbitrary program parts, ($G1$ is called a "guard", and $G1 \rightarrow S1$ a "guarded command") amounts to

- the execution of $S1$ if $G1$ holds, or the execution of $S2$ if $G2$ holds (if $G1 \wedge G2$ holds, an arbitrary choice is made between $S1$ and $S2$).
- if $\neg(G1 \vee G2)$ holds, the execution of the statement is suspended until $G1 \vee G2$ holds.

iii) $*[S]$ stands for "repeat S forever".

iv) $[G]$ where G is a Boolean, stands for $[G \rightarrow$ **skip**$]$, and thus for "wait until G holds". (Hence, "$[G]; S$" and $[G \rightarrow S]$ are equivalent.)

v) From ii) and iii), the operational description of the statement $*[[G_1 \rightarrow S_1 \mid \ldots \mid G_i \rightarrow S_i \mid \ldots]]$ is "repeat forever: wait until some G_i holds; execute an S_i for which G_i holds".

The implementation of the bar ($|$) when a non-deterministic choice has to be made is more difficult than when at most one guard is **true**. We shall therefore make the sets of guards as deterministic as possible. One form of non-determinacy is unavoidable when asynchronous signals may change the value of the guards while they are evaluated. This case presents the added complication of the metastability phenomenon, and the bar must then be implemented with an *arbiter*. Since it is difficult (if not impossible) to decide upon the necessity of arbitration at "compile time", we require that the programmer indicate whether a bar is an "arbitration bar" by using another symbol for this case: the thick bar $\|$.

Processes communicate with each other by communication actions on channels. A master M communicates with its server m via a channel

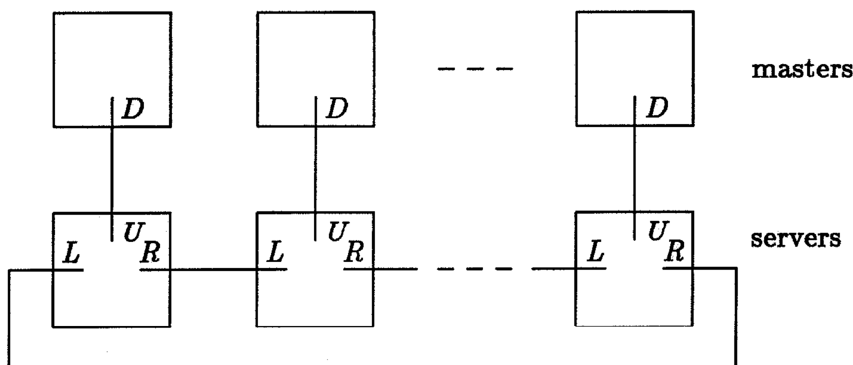


Figure 1

named D in M and U in m. A server m communicates with its right-hand neighbor mr by a channel named R in m and L in mr.

When no messages are transmitted, communication on a channel is reduced to synchronization signals. The name of the channel is then sufficient for identifying a communication action. For instance, the activity of a master relevant to our problem is described as:

$$M \equiv *[\ldots D;\ \text{"use resource"};\ D \ldots].$$

If two processes p and q share a channel named X in p and Y in q, at any time, the number of completed X-actions in p equals the number of completed Y-actions in q. In other words, the completion of the n-th X-action "coincides" with the completion of the n-th Y-action. If p reaches the n-th X-action before q reaches the n-th Y-action, the completion of X is suspended until q reaches Y. The X-action is then said to be *pending*. When q reaches Y, Y is *firable*.

A Boolean command on channels is used, called the ***probe***. In process p, the probe command $\overline{X}$ means "X is firable" or in other words, a "Y-action is pending in q". Hence $\overline{X} \rightarrow X$ guarantees that the X-action is not suspended. (For a more rigorous definition of the communication mechanism, see [3].)

2.2 The program

With the above programming notation, we can now give the first description of a server's algorithm. Each server owns a private Boolean b representing the privilege.

$$\begin{aligned} m \equiv *[[&\overline{U} \rightarrow [b \rightarrow \textbf{skip} \mid \neg b \rightarrow R];\ U;\ U;\ b\uparrow \\ \| &\overline{L} \rightarrow [b \rightarrow \textbf{skip} \mid \neg b \rightarrow R];\ L;\ b\downarrow \\ ]&]. \end{aligned}$$

(For a formal treatment of this algorithm and a correctness proof, see [4].)

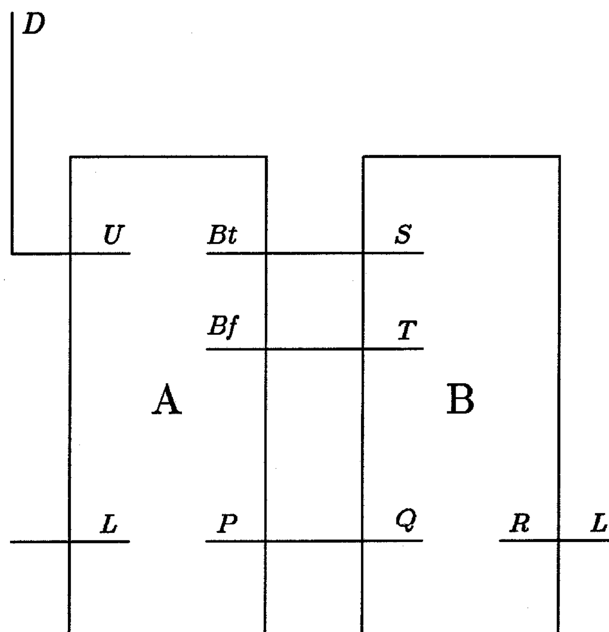


Figure 2

2.3 First decomposition

We distinguish two logical parts in a server's program: the "selector" A, which is the interface with the master and the left neighbor, and the "switch" B, which either transmits a request to the right or reflects it to the left, and which holds and updates the value of b. We therefore decompose process m into two communicating processes A and B. The decomposition is mechanically carried out by applying the following

Decomposition rule: *A process p containing an arbitrary program part S ($p \equiv \ldots S;\ \ldots\ S;\ \ldots$) is semantically equivalent to the two processes p1 and p2, where p1 is derived from p by replacing each occurrence of S by a communication action C on the newly introduced channel C, and $p2 \equiv *[[\overline{D} \rightarrow S;\ D]]$ with $p2.D = p1.C$.*

The communication between A and B is given by Fig. 2.

$$A \equiv *[[\overline{U} \rightarrow P;\ U;\ U;\ Bt \ \| \ \overline{L} \rightarrow P;\ L;\ Bf \ ]].$$

$$B \equiv *[[\overline{Q} \wedge b \rightarrow Q \ | \ \overline{Q} \wedge \neg b \rightarrow R;\ Q \ | \ \overline{S} \rightarrow b\uparrow;\ S \ | \ \overline{T} \rightarrow b\downarrow;\ T \ ]].$$

3. The "object code": operators and wires

3.1 Variables and wires

The object code is a collection of Boolean variables and operators

on those variables. A variable belongs exactly to one isochronic region. Because of the definition of isochronic regions, a wire inside an isochronic region represents a variable. Hence the symbols

$$\underline{x \quad\quad y} \qquad\qquad \underline{x \quad\quad y}$$
$$\qquad\qquad\qquad\quad |_{z}$$

inside an isochronic region mean $x \equiv y$, and $x \equiv y \wedge y \equiv z$ respectively. In the first case x and y, in the second case, x, y, and z are different names for the same variable. On the other hand, a wire between two isochronic regions—called a *non-isochronic wire*—is an operator. The main assumption on the physical behavior of a non-isochronic wire is the

Monotonicity property: *A monotonic change at one end of a non-isochronic wire is followed, if left undisturbed long enough, by the same monotonic change at the other end of the wire.*

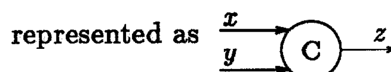
Because we make no assumption on the timing behavior or on the length of non-isochronic wire, in general we cannot guarantee that two consecutive changes at one end of a non-isochronic wire are followed by two corresponding changes at the other end of the same wire. In order to guarantee that all changes at one end of a wire are followed by the corresponding changes at the other end of the wire, we will restrict the use of non-isochronic wires to so-called four-phase handshaking protocols, i.e. as implementations of communication actions.

3.2 Operators

Apart from the wire-operator that will be described after introducing the handshaking protocol, the operators used are:

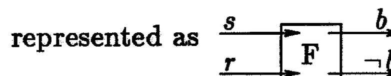
The C-element:

$$(x, y)\ \underline{C}\ z \equiv *[[x \wedge y \rightarrow z\uparrow \;|\; \neg x \wedge \neg y \rightarrow z\downarrow]].$$

represented as 

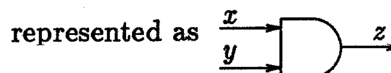
The set-reset flip-flop:

$$(s, r)\ \underline{F}\ b \equiv *[[s \rightarrow b\uparrow \;|\; r \rightarrow b\downarrow]].$$

represented as 

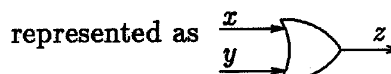
The "and":

$$(x, y)\ \triangle\ z \equiv *[[x \wedge y \rightarrow z\uparrow \;|\; \neg x \vee \neg y \rightarrow z\downarrow]].$$

represented as 

The "or":

$$(x, y)\ \underline{\vee}\ z \equiv *[[x \vee y \rightarrow z\uparrow \;|\; \neg x \wedge \neg y \rightarrow z\downarrow]].$$

represented as 

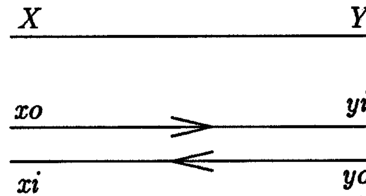


Figure 3

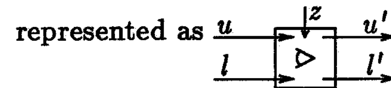
The inverter:

$$x \sqsupset z \equiv *[[x \rightarrow z\downarrow \;|\; \neg x \rightarrow z\uparrow]].$$

represented as

The arbiter:

$$(u, \ell, z)\ \underline{A}\ (u', \ell') \equiv *[[u \wedge \neg\ell' \wedge \neg z \rightarrow u'\uparrow \;|\; \ell \wedge \neg u' \wedge \neg z \rightarrow \ell'\uparrow \;|\; u' \wedge \neg u \rightarrow u'\downarrow \;|\; \ell' \wedge \neg\ell \rightarrow \ell'\downarrow]].$$

represented as 

In the arbiter, z is an inhibition variable that is used to postpone the firing of one of the first two commands until a sequence of actions caused by the previous firing of one of these two commands, is terminated. *A guarded command is called a production rule when the command contains no sequencing (no semicolon), as is the case for all operators described above.* The arbiter is the only operator able to make an arbitrary choice between two firable production rules (the conditions $u \wedge \neg\ell' \wedge \neg z$ and $\ell \wedge \neg u' \wedge \neg z$ can be **true** at the same time). (We assume that the arbiter makes a **fair** choice between the two commands, i.e. it is excluded that a command becomes firable infinitely often without being selected.) For all other operators, only one production rule may be firable at any time. This is only a problem for the flip-flop for which $\neg(s \wedge r)$ must hold at any time.

4. Handshaking

4.1 Four-phase handshaking

The channel (X, Y) shared by processes p and q is implemented by a pair of "directed wires": it is replaced in p by the output wire xo and the input wire xi, and in q by the output wire yo and the input wire yi (Fig. 3).

A matching pair of communication actions is always implemented as one output action denoted !, and one input action denoted ?. When no message is actually transmitted, an arbitrary choice is made unless the choice is dictated by the presence of probes. The communication actions on (X, Y) can be implemented either as X output $(X!)$ and Y input $(Y?)$:

$$X! \equiv xo\uparrow;\ [xi];\ xo\downarrow;\ [\neg xi] \qquad (1)$$
$$Y? \equiv [yi];\ yo\uparrow;\ [\neg yi];\ yo\downarrow \qquad (2)$$

or vice versa $(X?, Y!)$. Initially, all variables are **false**. A "probed" communication action $\overline{X} \rightarrow \ldots X$ must be implemented:

$$xi \rightarrow \ldots;\ xo\uparrow;\ [\neg xi];\ xo\downarrow. \qquad (3)$$

This implementation of probes forces the matching communication action to be implemented as an output. (Hence the two actions of a matching pair cannot be both probed.)

4.2 Basic properties

Several properties of the above handshaking protocol that play an important role in the compilation method are now given without proof.

Property 1: *The implementation of communication actions described in (1), (2), and (3) fulfills the semantics of communication actions and probe as given in 2.1.*

Property 2: *A wire between two isochronic regions that is used only for implementing a channel as described in 4.1, behaves as a so-called "wire operator" $\underline{W}$. The directed wire from x to y is the operator $x\ \underline{W}\ y$, with the semantics:*

$$x\ \underline{W}\ y \equiv *[[x \rightarrow y\uparrow \\ |\neg x \rightarrow y\downarrow \\ ]].$$

(In other words, the use of a non-isochronic wire in a handshaking protocol guarantees that any change at the input of the wire is effectively followed by the same change at the output of the wire.)

Property 3: *For the pair of wires $xo\ \underline{W}\ yi$ and $yo\ \underline{W}\ xi$, used together as in (1) and (2), and all variables* **false** *initially, the following sequence of transitions is guaranteed to occur if the system is deadlock-free:*

$$\{xo\uparrow,\ yi\uparrow,\ yo\uparrow,\ xi\uparrow,\ xo\downarrow,\ yi\downarrow,\ yo\downarrow,\ xi\downarrow\}^*.$$

Property 4: *Consider a program p containing a sequence of handshaking actions corresponding to the implementation of a communication action according to (1) or (2). Provided that the cyclic order of the four actions is respected, the last two actions can be inserted at any place in p without invalidating the semantics of the communication involved. However, modifying the order of these two actions relatively to other actions of p may introduce deadlock.*

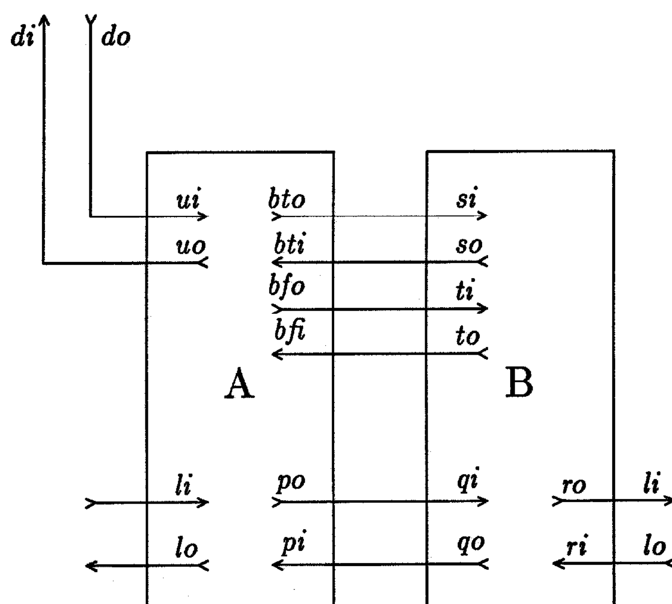


Figure 4

5. The "compilation" procedure

The first step of the "compilation" consists in replacing each communication action by its four-phase handshaking implementation. This compilation step is called the *handshaking expansion* of the program.

5.1 Handshaking expansion

The expansion of the communication structure according to the handshaking protocol of 4.1 is shown in Fig. 4.

In the programs of M, A, and B we are now going to replace the communication actions by their handshaking implementation. When this mechanical task is completed, and possibly after clerical simplifications, some transformations are applied based on Property 4.

We perform a simple optimization on the expansion of A and consequently on the expansion of M: to suppress one of the two consecutive U-expansions. The expansions of M and A become:

$$M \equiv *[\ldots;\ do\uparrow;\ [di];\ \text{"use resource"};\ do\downarrow;\ [\neg di]\ldots]. \tag{4}$$

$$\left.\begin{array}{l} A \equiv *[[ui \rightarrow po\uparrow;\ [pi];\ po\downarrow;\ [\neg pi];\ uo\uparrow;\ [\neg ui]; \\ \qquad\qquad uo\downarrow;\ bto\uparrow;\ [bti];\ bto\downarrow;\ [\neg bti] \\ \quad [\!]\ \ell i \rightarrow po\uparrow;\ [pi];\ po\downarrow;\ [\neg pi];\ \ell o\uparrow;\ [\neg \ell i]; \\ \qquad\qquad \ell o\downarrow;\ bfo\uparrow;\ [bfi];\ bfo\downarrow;\ [\neg bfi] \\ \quad]]. \end{array}\right\} \tag{5}$$

The straightforward expansion of B gives:

$$
\left.\begin{array}{l}
B \equiv *[[qi \wedge b \rightarrow qo\uparrow;\ [\neg qi];\ qo\downarrow \\
\qquad |qi \wedge \neg b \rightarrow ro\uparrow;\ [ri];\ ro\downarrow;\ [\neg ri];\ qo\uparrow;\ [\neg qi];\ qo\downarrow \\
\qquad |si \rightarrow b\uparrow;\ so\uparrow;\ [\neg si];\ so\downarrow \\
\qquad |ti \rightarrow b\downarrow;\ to\uparrow;\ [\neg ti];\ to\downarrow \\
\qquad]].
\end{array}\right\} \quad (6)
$$

5.2 Production rules expansion

The next step is to compile the handshaking expansions of the programs into sets of production rules from which all explicit sequencing has been removed. By matching those production rules to those describing the semantics of each operator, the programs can be identified with networks of operators. Consider the compilation of B.

5.3 Compilation of B

The right-hand side of each production rule in the next compilation of B will contain exactly one transition of (6), i.e. one of the actions $qo\uparrow$, $qo\downarrow$, $ro\uparrow$, $ro\downarrow$, etc ... Since there are twelve occurrences of these transitions in (6), there will be twelve production rules. The problem is to define the guard of each rule such that the sequence of firings of these rules is equivalent to the execution of (6). In determining these guards we take into account that

i) all handshaking variables are initialized to **false**,

ii) Property 3 induces sequencing among set of transitions,

iii) an upward transition ($x\uparrow$) can be guarded by negated variables only, if it is the first ("spontaneous") transition of a sequence. (Obviously, only the first transition in a master's sequence is of this type. None of the servers' transitions is.)

The production rules for the first, third, and fourth guarded commands of (6) can be derived easily. For instance, for the first guarded command we obtain the rules

$$
\begin{array}{c}
qi \wedge b \rightarrow qo\uparrow \\
\neg qi \wedge b \rightarrow qo\downarrow .
\end{array}
$$

But we have some difficulty in determining the guard of $qo\uparrow$ in the second guarded command, since the only variable with value **true** as a precondition of $qo\uparrow$ is qi. In order to define uniquely the state in which the transition $qo\uparrow$ is to take place we can, of course, introduce a state variable. But the implementation of a state variable requires in general a C-element or a flip-flop, and in this solution we choose to minimize the number of those "expensive" state-holding elements.

Instead, we are going to transform the second guarded command according to the transformation allowed by Property 4: we postpone the actions $ro\downarrow$; $[\neg ri]$ until after $[\neg qi]$. The command now becomes:

$$qi \wedge \neg b \rightarrow ro\uparrow;\ [ri];\ qo\uparrow;\ [\neg qi];\ ro\downarrow;\ [\neg ri];\ qo\downarrow . \qquad (7)$$

We have to check that the insertion of $qo\uparrow;\ [\neg qi]$ between the two halves of the R-protocol sequence does not introduce deadlock. This is easily done by observing that when B is suspended at $[\neg qi]$, the next action of A is $po\downarrow$ and thus there cannot be deadlock. With this transformation, the compilation of (7) is possible without extra state variables. Hence the complete production rule compilation of B:

$$
\begin{array}{llr}
B \equiv *[& [qi \wedge b \rightarrow qo\uparrow & (B1)\\
 & |\neg qi \wedge b \rightarrow qo\downarrow & (B2)\\
 & |qi \wedge \neg b \rightarrow ro\uparrow & (B3)\\
 & |ri \rightarrow qo\uparrow & (B4)\\
 & |\neg qi \wedge \neg b \rightarrow ro\downarrow & (B5)\\
 & |\neg ri \rightarrow qo\downarrow & (B6)\\
 & |si \rightarrow b\uparrow & (B7)\\
 & |b \wedge si \rightarrow so\uparrow & (B8)\\
 & |\neg si \rightarrow so\downarrow & (B9)\\
 & |ti \rightarrow b\downarrow & (B10)\\
 & |\neg b \wedge ti \rightarrow to\uparrow & (B11)\\
 & |\neg ti \rightarrow to\downarrow & (B12)\\
 &]]. &
\end{array}
$$

We leave it to the reader to check that the execution of the above program started with all handshaking variables false and an arbitrary value of b, corresponds to the execution of (6) started in the same state. (The ordering between handshaking transitions stated in Property 3 has to be taken into account.)

We are now able to implement sets (pairs, actually) of rules with operators on variables of B. $B1$ and $B2$ correspond to $(qi, b) \underline{\wedge}\, qo$. $B4$ and $B6$ correspond to $ri = qo$ (isochronic wire or super-buffer). Hence, we implement those four rules as:

$$((qi \wedge b), ri)\ \underline{\vee}\ qo .$$

The implementation of the other rules is straightforward.

$B3$ and $B5$: $(qi, \neg b)\ \underline{\wedge}\ ro$.
$B7$ and $B10$: $(si, ti)\ \underline{F}\ b$, since $\neg(si \wedge ti)$ holds at any time.
$B8$ and $B9$: $(b, si)\ \underline{\wedge}\ so$.
$B11$ and $B12$: $(\neg b, ti)\ \underline{\wedge}\ to$.

This network is represented in Fig. 5.

5.4 Compilation of A

The compilation of A is slightly more complicated since it requires the use of an arbiter. Let us postpone the arbitration issue for a while and concentrate on the production rule implementation of the first guarded

Figure 5

command of (4). (Since the two commands are symmetrical, the implementation of the second one is identical.) We meet the same difficulty for the production rules of the upward transitions $uo\uparrow$ and $bto\uparrow$ as for $qo\uparrow$ in the compilation of B. And we apply the same technique to resolve it: applying the transformation allowed by Property 4, we postpone the actions $po\downarrow$; $[\neg pi]$ until after $[\neg ui]$, and the action $uo\downarrow$ until after $[bti]$. The code of A now becomes:

$$\left.\begin{array}{l} *[[ui \rightarrow po\uparrow;\ [pi];\ uo\uparrow;\ [\neg ui];\ po\downarrow;\ [\neg pi]; \\ \qquad\qquad\qquad bto\uparrow;\ [bti];\ uo\downarrow;\ bto\downarrow;\ [\neg bti] \\ \;\,\|\ell i \rightarrow po\uparrow;\ [pi];\ \ell o\uparrow;\ [\neg \ell i];\ po\downarrow;\ [\neg pi]; \\ \qquad\qquad\qquad bfo\uparrow;\ [bfi];\ \ell o\downarrow;\ bfo\downarrow;\ [\neg bfi] \\ \;\,]]. \end{array}\right\} \quad (5')$$

The verification that those transformations do not introduce deadlock is hardly more difficult than in the previous case, and is omitted.

In order to implement the thick bar in (5′), we have to introduce an arbiter $(ui, \ell i, z)\ \underline{A}\ (u', \ell')$. The next coding of A consists of the composition of the guarded commands of the arbiter and the guarded commands of (5′) in which u' and ℓ' have been substituted for ui and ℓi:

$$
\left.\begin{array}{l}
*[[ui \wedge \neg \ell' \wedge \neg z \rightarrow u' \uparrow \\
\;|\ell i \wedge \neg u' \wedge \neg z \rightarrow \ell' \uparrow \\
\;|u' \wedge \neg ui \rightarrow u' \downarrow \\
\;|\ell' \wedge \neg \ell i \rightarrow \ell' \downarrow \\
\;|u' \rightarrow po \uparrow;\ [pi];\ uo \uparrow;\ [\neg u'];\ po \downarrow; \\
\qquad\qquad [\neg pi];\ bto \uparrow;\ [bti];\ uo \downarrow;\ bto \downarrow;\ [\neg bti] \\
\;|\ell' \rightarrow po \uparrow;\ [pi];\ \ell o \uparrow;\ [\neg \ell'];\ po \downarrow; \\
\qquad\qquad [\neg pi];\ bfo \uparrow;\ [bfi];\ \ell o \downarrow;\ bfo \downarrow;\ [\neg bfi] \\
]].
\end{array}\right\} \quad (8)
$$

For the inhibition variable z (z is an internal variable of A), we choose:

$$z = uo \vee \ell o \vee bti \vee bfi \,.$$

With this definition of z, it is easy to verify that the execution of (8) is equivalent to the execution of (5′) started in the same state. The transformation of the 5th and 6th guarded commands of (8) into production rules is now straightforward. For the 5th guarded command, we get:

$$
\begin{array}{rcll}
u' & \rightarrow & po \uparrow & (A1) \\
u' \wedge \ pi & \rightarrow & uo \uparrow & (A2) \\
\neg u' & \rightarrow & po \downarrow & (A3) \\
\neg pi \wedge uo & \rightarrow & bto \uparrow & (A4) \\
bti & \rightarrow & uo \downarrow & (A5) \\
\neg uo & \rightarrow & bto \downarrow \,. & (A6)
\end{array}
$$

Observe that since $bti \wedge bfi \Rightarrow z$, our choice of the inhibition variable takes care of the implementation of the last transitions $[\neg bti]$ and $[\neg bfi]$. The implementation in terms of operators is as follows.

For $A1$ and $A3$: $u' = po$. For $A4$ and $A6$: $(\neg pi, uo) \,\underline{\wedge}\, bto$, since $\neg pi$ holds as precondition of $A6$. For $A2$ and $A5$, we strengthen the guards as:

$$
\begin{array}{rcll}
u' \wedge pi \wedge \neg bti & \rightarrow & uo \uparrow & (A2) \\
\neg(u' \wedge \ pi) \wedge bti & \rightarrow & uo \downarrow \,, & (A5)
\end{array}
$$

which admits the implementation $((u' \wedge pi),\ \neg bti)\ \underline{C}\ uo$.

Combining this set of operators with those of the other guards, we obtain the complete set:

$$
\begin{array}{ll}
(ui, \ell i, z)\ \underline{A}\ (u', \ell') & \\
(uo, \ell o, bti, bfi)\ \underline{\vee}\ z & \text{(generalized or)} \\
(u', \ell')\ \underline{\vee}\ po & \\
(\neg pi, uo)\ \underline{\wedge}\ bto & \\
(\neg pi, \ell o)\ \underline{\wedge}\ bfo & \\
((u' \wedge pi), \neg bti)\ \underline{C}\ uo & \\
((\ell' \wedge pi), \neg bfi)\ \underline{C}\ \ell o \,. &
\end{array}
$$

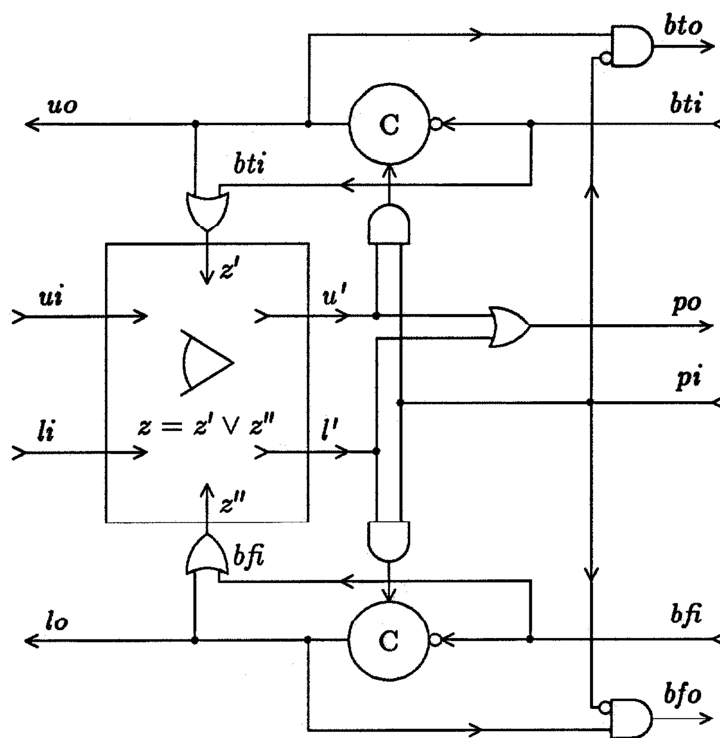


Figure 6

This network is represented in Fig. 6. The complete circuit is represented in Fig. 7.

6. Conclusion

We have described a method for implementing a high-level concurrent algorithm (a set of communicating processes) as a network of digital operators. We have chosen to illustrate the method with an example: the design of a self-timed circuit for distributed mutual exclusion. Although the example chosen is far from trivial, we arrive at the solution by a series of systematic, semantics-preserving, transformations that we have compared to compiling.

The proofs that the transformations preserve the semantics of the algorithms rely on four properties of the four-phase handshaking protocol with which the communication primitives are implemented. Although the proofs of these properties have been omitted, the reader should have no difficulty in being convinced of their correctness, and thus of the correctness of the transformations performed.

The main step in the translation process is the transformation of a strictly sequential algorithm—the handshaking expansion of a communicating process—into a set of production rules from which all explicit

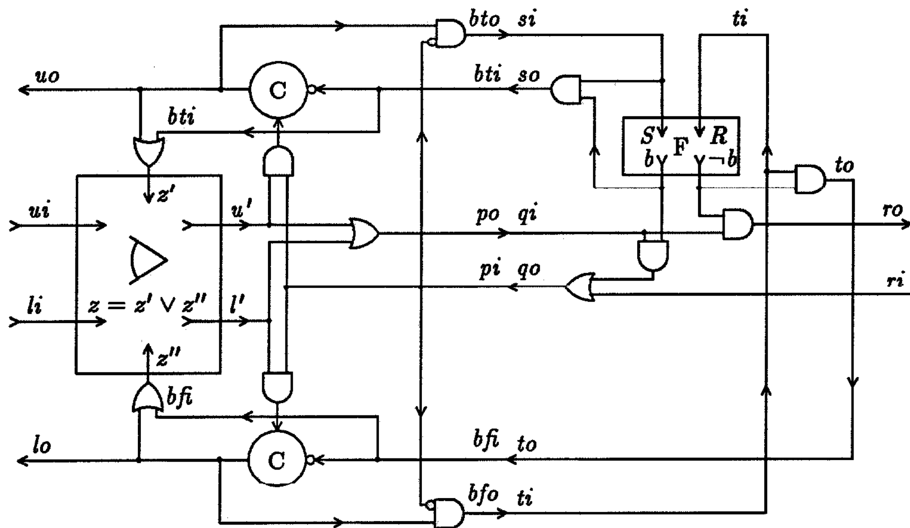


Figure 7

sequencing has been removed. Yet, we guarantee that the rules are fired one at a time in a sequence equivalent to the sequential execution of the original program. We have seen that for the production rule of certain transitions—upward transitions following a complete handshaking sequence—it is impossible, without transformation, to determine a guard that enforces the required sequencing. We have mentioned that this problem can be solved in a quite mechanical way by introducing a state variable that identifies this state uniquely, i.e. by introducing a C-element or a flip–flop.

Although, in this case, the number of extra C-elements thus introduced would be moderate (2 or 3), we aimed at minimizing the number of state-holding elements, and have opted for another—less mechanical—method, consisting of "moving" the second half of some handshaking sequences. This method requires a little care concerning deadlock, but can lead to quite interesting solutions. We believe that, in the solution obtained, the number of state-holding elements is minimal.

Observe that since only one production rule is fired at a time the occurrence of hazards is excluded.

Acknowledgment

Acknowledgment is due to Chuck Seitz for several helpful suggestions on the choice of operators and the structure of arbiters [7] and for invaluable comments on previous versions of the paper, and to Jan van

de Snepscheut and Martin Rem for their comments on a previous solution. Special thanks to our TEXperts Calvin Jackson and Wen-King Su for their crucial help.

The research described in this paper was sponsored in part by the Defense Advanced Research Projects Agency, ARPA Order number 3771, and was monitored by the Office of Naval Research under contract number N00014-79-C-0597.

References

[1] Chaney, T.J. and C.E. Molnar, "Anomalous Behavior of Synchronizer and Arbiter Circuits", *IEEE Transactions on Computers*, vol. C-22, no.4, April 1973, pp. 421-422

[2] Dijkstra, Edsger W., *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs NJ (1976)

[3] Martin, A.J., "The Probe: An Addition to Communication Primitives", to appear in *Information Processing Letters* (1985), also Caltech Computer Science Report 5124:TR:84

[4] Martin, A.J., "Distributed Mutual Exclusion on a Ring of Processes" Computer Science, Caltech, 5080:TR:83

[5] Mead, C. and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading MA (1980)

[6] Seitz, C.L., "System Timing", Chapter 7 in Mead & Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, MA (1980)

[7] Seitz, C.L., Computer Science, Caltech, Lecture Notes (1983)