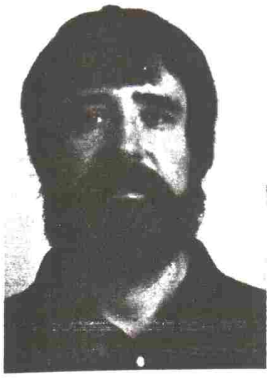
Distributed Computing (1986) 1:150–166

DISTRIBUTED COMPUTING

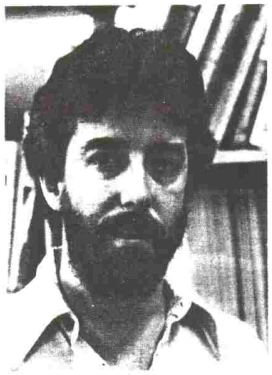
Compiling path expressions into VLSI circuits

T.S. Anantharaman, E.M. Clarke, M.J. Foster[1], B. Mishra
Carnegie-Mellon University Pittsburgh, Pennsylvania 15213, USA

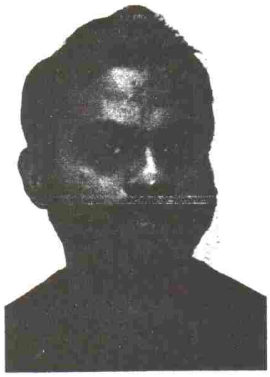
Thomas S. Anantharaman *received a B. Tech degree in Electronics Engineering from the Banaras Hindu University, Varanasi (India), in 1982, and has been a graduate student in the computer science department at Carnegie-Mellon University since then.*



Edmund M. Clarke, Jr. *received a B.A. degree in mathematics from the University of Virginia, Charlottesville, in 1967, an M.A. degree in mathematics from Duke University, Curham N.C., in 1968 and a Ph.D. degree in computer science from Cornell University, Ithaca, NY in 1976. After leaving Cornell, he taught in the department of computer science, Duke University, for two years. In 1978 he moved to Harvard University where he was an assistant professor of computer science in the Division of Applied Sciences for four years. He is currently an associate professor in the computer science department at Carnegie Mellon University. His interests include distributed systems, VLSI design, programming language semantics, and theory of computation. Dr. Clarke is a member of the Association for Computing Machinery, Sigma Xi, Phi Beta Kappa, and is on the editorial board of Distributed Computing.*



Michael J. Foster *received the B.S. degree in mathematics from the Massachusetts Institute of Technology, Cambridge, in 1973, and the Ph.D. degree in computer science from Carnegie-Mellon University, Pittsburgh, in 1984. From 1973 to 1977 he worked at Tracor-Northern and Princeton Gamma-Tech. Since July 1984 he has been an Assistant Professor of Computer Science at Columbia University, New York City. His research and teaching interests include VLSI design, algorithms, and computer architecture. Foster used to believe that biographies of this type were written by editors who knew all of the researchers in their field, but has now learned to refer to himself in the third person. He is a member of IEEE, ACM, and ΣΞ.*



B. Mishra *received a B. Tech degree from the Indian Institute of Technology, M.S. and Ph.D. degrees, both from Carnegie-Mellon University. He will be joining New York University in the fall of 1985.*

1 *Current address:* Department of Computer Science, Columbia University, New York, NY 10027, USA

This research was partially supported by NSF Grant MCS-82-16706, and the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539

Abstract. Path expressions were originally proposed by Campbell and Habermann [2] as a mechanism for process synchronization at the monitor level in software. Not surprisingly, they also provide a useful notation for specifying the behavior of asynchronous circuits. Motivated by these po-

tential applications we investigate how to directly translate path expressions into hardware. Our implementation is complicated in the case of multiple path expressions by the need for synchronization on event names that are common to more than one path. Moreover, since events are inherently asynchronous in our model, all of our circuits must be self-timed. Nevertheless, the circuits produced by our construction have area proportional to $N \cdot \log(N)$ where N is the total length of the multiple path expression under consideration. This bound holds regardless of the number of individual paths or the degree of synchronization between paths. Furthermore, if the structure of the path expression allows partitioning, the circuit can be laid out in a distributed fashion without additional area overhead.

Key words: Silicon compilation – Path expressions – Process synchronization

1 Introduction

As the boundary between software and hardware grows less and less distinct, it becomes increasingly important to investigate methods of directly implementing various programming language features in hardware. Since many of the problems in interfacing hardware devices involve some form of process synchronization, language features for synchronization deserve considerable attention in such investigations. In this paper we consider the problem of directly implementing path expressions as self-timed VLSI circuits. Path expressions were originally proposed by Campbell and Habermann [2] for restricting access by other processes to the procedures of a monitor. For example, the simple readers and writers problem with two reader processes and a single writer process is solved by the following multiple path expression:

path $R_1 + W$ **end,**

path $R_2 + W$ **end.**

The first path expression prohibits a read operation by the first process from occurring at the same time as a write operation. The second path expression enforces a similar restriction on the behavior of the second reader process. In a computation under control of the multiple path expression, the two read operations may occur simultaneously, but a read and write operation cannot occur at the same time.

A *simple path expression* is a regular expression with an outermost Kleene star. The only operators permitted in the regular expression are (in order of precedence) "*", ";" , and " + ". The "*" operator is the Kleene star, ";" is the sequencing operator, and " + " represents exclusive choice. Operands are event names from some set of events Σ that we will assume to be fixed in this paper. The outermost Kleene star is usually represented by the delimiting keyword **path**...**end**. Thus (a)* would be represented as **path** a **end**. Roughly the sequence of events allowed by a simple path expression must correspond to the sequences in the language of the regular expression.

A *multiple path expression* is a set of simple path expressions. As we will see shortly, each additional simple path expression further constrains the order in which events can occur. However, we cannot simply take as our semantics for multiple path expressions the intersection of the languages corresponding to the individual path expressions; two events whose order is not explicitly restricted by one of the simple path expressions may be concurrent. For example, in the multiple path expression for the readers and writers problem discussed earlier the two read events R_1 and R_2 may occur simultaneously.

Path expressions are useful for process synchronization for two reasons: First, the close relationship between path expressions and regular expressions simplifies the task of writing and reasoning about programs which use this synchronization mechanism. Secondly, the synchronization in many concurrent programs is finite state and thus, can be adequately described by regular expressions. For precisely the same reasons, path expressions are useful for controlling the behavior of complicated asynchronous circuits. The readers and writers example above could equally well describe a simple bus arbitration scheme. In fact, the finite-state assumption may be even more reasonable at the hardware level than at the monitor level.

Which brings us to the topic of this paper: What is the best way to translate path expressions into circuits? Lauer and Campbell have shown how to compile path expressions into Petri nets [7], and Patil has shown how to implement Petri nets as circuits by using a PLA-like device called an asynchronous logic array [13]. Thus, an obvious method for compiling path expressions into circuits would be to first translate the path expression into a Petri net and then to implement the Petri net as a circuit using an asynchronous logic array. However, careful examination of Lauer and Campbell's scheme shows that a multiple path expression consisting of M paths each of length K can result in a Petri net with K^M places. Thus, the naive approach will in general be infeasible

if the number of individual paths in multiple path expression is large.

For the case of a path expression with a single path their scheme does result in Petri net which is comparable in size to the path expression. However, direct implementation of such a net using Patil's ideas may still result in a circuit with an unacceptably large area. An asynchronous logic array for a Petri net with P places and T transitions will have area proportional to $P \cdot T$ regardless of the number of arcs in the net. Since the nets obtained from path expressions tend to have sparse edge sets, this quadratic behavior may waste significant chip area.

Perhaps, the work that is closest to ours is due to Li and Lauer [10] who do indeed implement path expressions in VLSI. However, their circuits differ significantly from ours; in particular, their circuits are synchronous, and synchronization with the external world (which is, of course, inherently asynchronous) is not considered. (This means that the entire circuit, not just the synchronization, must be described using path expressions.) Furthermore, their circuits use PLA's that result in an area complexity of $O(N^2)$. Rem [15] has investigated the use of a hierarchically structured path expression-like language for specifying CMOS circuits. Although he does show how certain specifications can be translated into circuits, he does not describe how to handle synchronization or give a general layout algorithm that produces area efficient circuits.

In contrast, the circuits produced by the construction described in this paper have area proportional to $N \cdot \log(N)$ where N is the total length of the multiple path expression under consideration. Furthermore, this bound holds regardless of the number of individual paths or the degree of synchronization between paths. As in [4] and [5] the basic idea is to generate circuits for which the underlying graph structure has a constant separator theorem [8]. For path expressions with a single path the techniques used by [4] and [5] can be adapted without great difficulty. For multiple paths with common event names, however, the construction is not straightforward, because of the potential need for synchronization at many different points on each individual path. Moreover, the actual circuits that we use must be much more complicated than the synchronous ones used in ([4], [5]). Since events are inherently asynchronous in our model, all of our circuits must be self-timed and the use of special circuit design techniques is required to correctly capture the semantics of path expressions.

The paper is organized as follows: A formal semantics for path expressions in terms of partially ordered multisets [14] is given in Sect. 2. In Sect. 3, 4, and 5 we give a hierarchical description of our scheme for implementing path expressions as circuits. In Sect. 3 we first describe how the complete circuit interfaces with the external world. We then show how to build a *synchronizer* that coordinates the behavior of the circuits for the individual path expressions in a multiple path expression. In Sect. 4 we describe a circuit for implementing single path expressions which we call a *sequencer*. In Sect. 5 we show how the arbiter circuit used in Sect. 3 can be implemented. We also argue that these circuits are correct and can be laid out efficiently. The conclusion in Sect. 6 discusses the feasibility of our implementation and the possibility of extending it to other synchronization mechanisms like those used in CCS and CSP.

2 The semantics of path expressions

In this section we give a simple but formal semantics for path expressions in terms of partially ordered multisets (*pomsets*) of events [14]. An alternative semantics in terms of Petri Nets is given by Lauer and Campbell in [7]. A a pomset may be regarded as a generalization of a sequence in which certain elements are permitted to be concurrent; this is why the concept is useful in modeling systems where several events may occur simultaneously.

Definition 1. A *partially ordered multiset* (pomset) over Σ is a triple $(Q, \leq, F)$ where $(Q, \leq)$ is a partially ordered set and F is a function which maps Q into Σ. □

An example of a pomset is shown in Fig. 1. We use subscripts to distinguish different elements of Q that map to the same element of Σ. In this case $Q = (A_1, A_2, A_3, B_1, B_2, B_3, C_1, C_2, C_3)$ and $\Sigma = (A, B, C)$. Note that we could have alternatively defined a pomset as a directed acyclic graph in which each node is labeled with some element of Σ.

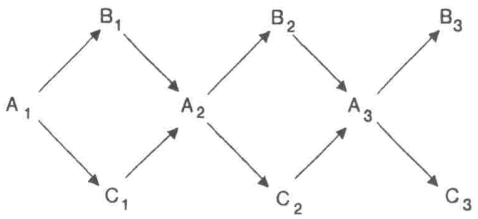


Fig. 1. An example pomset

If the ordering relation of a pomset P over Σ is a total order, then we can naturally associate a sequence of elements of Σ with P; we will use S(P) to denote this sequence.

Definition 2. If $P=(Q, \leq, F)$ is a pomset over Σ and $\Sigma_1 \subseteq \Sigma$, then the *restriction* of P to Σ_1 is the pomset $P|_{\Sigma_1}=(Q_1, \leq_1, F_1)$ where $Q_1=\{d \in Q | F(d) \in \Sigma_1\}$ and $\leq_1, F_1$ are restrictions of $\leq$, F to Q_1, respectively. □

If P is a totally ordered pomset over Σ and $\Sigma_1 \subseteq \Sigma$, then $S(P|_{\Sigma_1})$ is just the *subsequence* of S(P) obtained by deleting all of those elements of Σ which are not in Σ_1. If R is an ordinary regular expression over Σ, then $\Sigma_R \subseteq \Sigma$ will be the set of symbols of Σ that actually appear in R and $L_R \subseteq \Sigma_R^*$ will be the regular language which corresponds to R.

Definition 3. Let Σ be a finite set. A *trace* over Σ is a pomset $T=(Q, \leq, F)$ over Σ such that every infinite chain of the partially ordered set $(Q, \leq)$ is an ω-sequence[1]. We say that $i \in Q$ is an *instance* of an event $e \in \Sigma$ if $F(i)=e$. An instance i_1 of event e_1 *precedes* an instance i_2 of event e_2 if i_1 precedes i_2 in the partial order $\leq$. An instance i_1 of event e_1 is *concurrent* with an instance i_2 of event e_2, if neither instance precedes the other. □

In the example above A_1 precedes A_2, but B_1 and C_1 are concurrent.

Definition 4. Let R be a simple path expression with event set Σ_R. A trace T is *consistent with* R iff $T|_{\Sigma_R}$ is totally ordered and every finite prefix of $S(T|_{\Sigma_R})$ is a prefix of some sequence in L_R, the language of regular expression R. If M is a multiple path expression, then a trace T is *consistent with* M iff it is consistent with each simple path expression R in M. $Tr_\Sigma(M)$ is the set of all traces which are consistent with M. □

Consider, for example, the multiple path expression M:

path A; B **end,**

path A; C **end.**

with $\Sigma=\{A, B, C\}$. It is easy to see that the trace in Fig. 1 is consistent with each of the simple path expressions in M and hence is in $Tr_\Sigma(M)$.

1 In absence of fairness, finite sequences are sufficient. In order to talk about fairness however, we require infinite sequences. An ω-sequence is the shortest infinite sequence that captures the semantics of fairness and has the advantage that all of its prefixes are finite

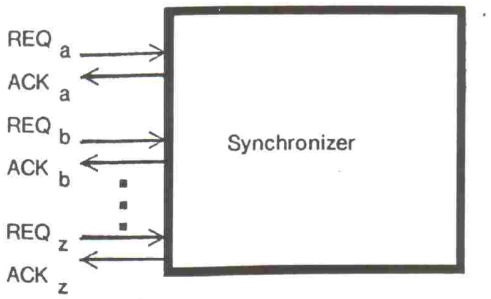


Fig. 2. A synchronizer

3 Synchronizers for multiple path expressions

This section describes our implementation of synchronizers for multiple path expressions. Figure 2 illustrates the interface between a synchronizer and the external world. Each event *e* is associated with a request line REQ_e and acknowledge line ACK_e. The synchronizer cooperates with the external world to ensure that these request and acknowledge lines follow a 4-cycle protocol:

1. The external world raises REQ_e to indicate that it would like to proceed with event *e*.
2. The synchronizer raises ACK_e to allow the external world to proceed with event *e*.
3. The external world lowers REQ_e, signifying completion of event *e*.
4. The synchronizer lowers ACK_e, signifying the end of the cycle and permission to begin a new one.

In this implementation, an event will occur during the period between cycles 2 and 3 in this protocol, where both REQ and ACK are high. Thus, multiple occurrences of any event *e* are non-overlapping in time, since any two occurrences are separated by the lowering of ACK and the raising of REQ.

In a distributed system each of the devices in the system would be a *client* of the synchronizer; only a subset of the REQ and ACK lines would go to each device. Before performing an action, each client would request permission from the synchronizer and wait until permission was granted. In this way, harmonious cooperation could be ensured with only a small amount of inter-device communication. Because of the symmetric nature of the protocol any *client* could act either as a master or a slave relative to other *clients*. A slave would always assert all REQ's and wait for a response through the ACK's telling it what to do, whereas a master would assert REQ's only for those events it wishes to proceed with and use the ACK's only to get its timing correct.

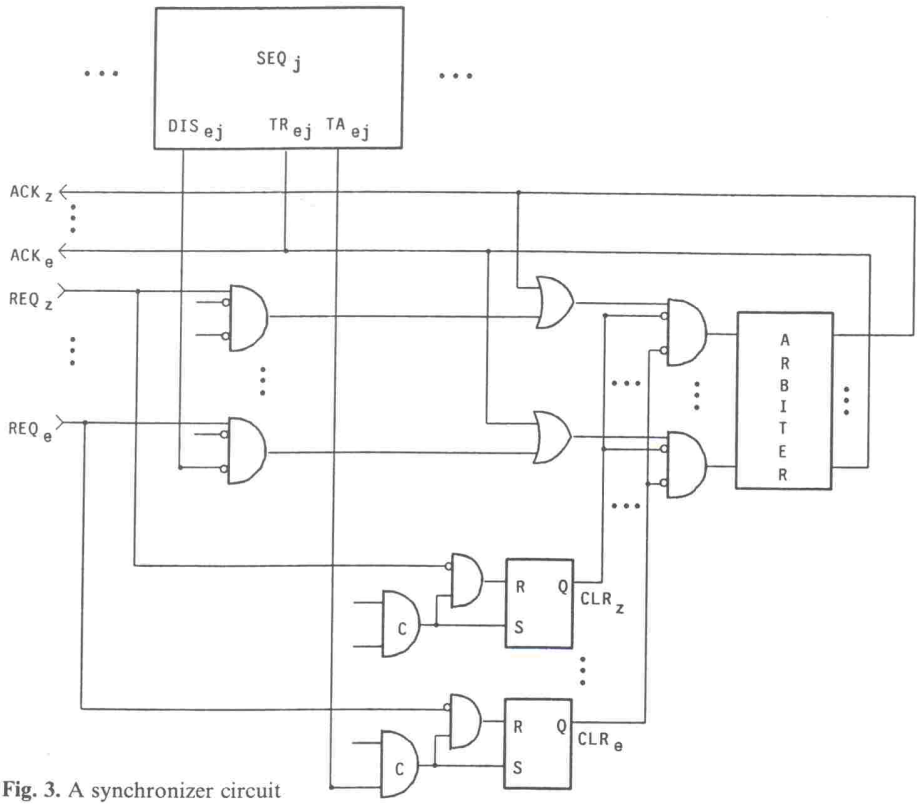


Fig. 3. A synchronizer circuit

An overview of a synchronizer circuit is shown in Fig. 3. The circuit shown is self timed but not delay independent as it makes certain assumptions about gate delays which will be described later. Some of the building blocks in the circuit are described below.

The C gate in Fig. 3 is a Muller C-element; the output of a C-element remains low until all inputs are high and thereafter remains high until all inputs are low again. Its behavior then cycles. For an implementation see [16].

The arbiter in Fig. 3 enforces pairwise mutual exclusion over the outputs corresponding to pairs of events which occur in the same path expression. In addition to enforcing mutual exclusion the arbiter tries to raise any output whose input is high. Many implementations of arbiters will have metastable states during which fewer signals than possible may be high at the output. Despite the metastable states, however, once an output signal has been raised, it must remain high as long as the corresponding input remains high. The implementation of such an arbiter is discussed in detail in Sect. 5.

Each sequencer block in Fig. 3 ensures that the sequence of events satisfies one of the simple path expressions that comprise the multiple path expression, and will be described in the next section. The synchronizer circuit contains one sequencer for each simple path expression, so that each simple path expression is satisfied by an execution event trace. For each event e that appears in a simple path, the corresponding sequencer has three connections: a request TR_e, an acknowledge TA_e, and a disable DIS_e. Events are sequenced by executing a 4-cycle protocol over one pair of the TR/TA lines. The DIS outputs of the sequencer are only valid between these cycles (when all TR and TA are low), and indicate which events would violate the simple path. The synchronizer will not initiate a cycle for any event whose DIS line is high.

We now describe how the components of the circuit are interconnected. Refer to Fig. 3. Let SEQ_e denote the set of sequencers for simple paths that contain event e. Every sequencer in SEQ_e has its DIS_e signal connected to a NOR gate for e, its TA_e signal connected to a C gate for e, and its TR_e signal connected to ACK_e. The output of the latch at the end of the C gate for e, which is labeled CLR_e, is connected to each of the NOR gates in front of the arbiter which corresponds to event e or to some event mutually exclusive to e.

Notice that there is no intrinsic need for the synchronizer to be centralized as long as the constraints themselves do not require it. Whenever the

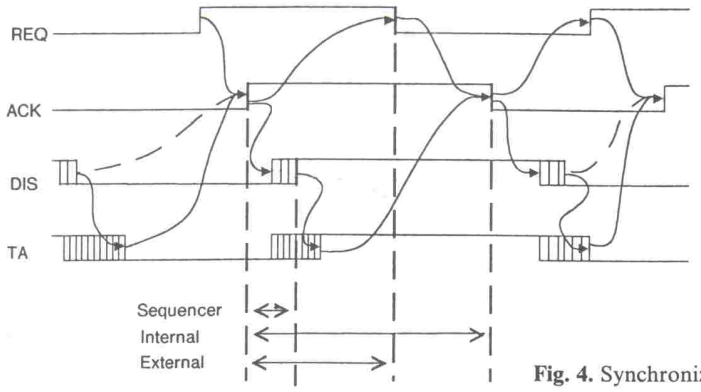


Fig. 4. Synchronizer timing

multiple path expression can be partioned into disjoint sets of paths so that paths in different sets do not refer to the same event, then each set can be implemented as a circuit independently of the others.

The following is an informal description of how the circuit works. The circuit behaves as shown in the timing diagram in Fig. 4. When REQ_e is raised, event e is not allowed to proceed unless each sequencer in SEQ_e signals that at least one e type transition is enabled by negating DIS_e. Once this happens IN_e is raised, provided no mutually exclusive event is executing the second half of its cycle (and hence has its CLR high). If the arbiter decides in favor of some other pending event mutually exclusive to e, the above process repeats until e again gets a chance at the arbiter. Otherwise ACK_e will be raised and latched by the NOR gate arrangement in front of the arbiter. At this point the external world may proceed with event e. Simultaneously each sequencer in SEQ_e will find TR_e high and after some time raise TA_e. When all sequencers in SEQ_e have raised TA_e and the external world acknowledges completion of event e by lowering REQ_e, CRL_e will be raised. This causes ACK_e to be lowered. Each sequencer in SEQ_e will find TR_e low and after some time lower TA_e. When all such sequencers are done, CRL_E is lowered, and the cycle is completed.

To formally establish the correctness of our circuit, we must establish two things: First, we must show that the circuit allows only semantically correct event traces; second, that the circuit will allow any semantically correct event trace for some behavior of the external world. These properties of the circuit are often called *safeness* and *liveness* respectively. A third important property, *fairness*, is dealt with in a separate section. Our proof will make use of properties of the various circuit components shown in Fig. 3. We list the most important of these properties as propositions, namely those relating to the sequencer, the arbiter, and the external world. Properties of other circuit components such as SR Flip-Flops, NOR gates, etc., are assumed to be well known and are used without further discussion. The proof also makes certain assumptions about the delays of the components:

1. The delay of the main NOR gate plus the 2-input OR gate is less than that of the main Muller-C element plus the SR Flip-Flop.
2. The maximum variation in delay for the NOR gates in front of the arbiter is less than the minimum delay of the arbiter.

We begin by introducing some notation that will be needed in the proof. Let the sequencers be denoted by $\text{SEQ}_1 \ldots \text{SEQ}_p$ corresponding to the path expressions $R1 \ldots Rp \in M$, and let $\Sigma_{R1} \ldots \Sigma_{Rp}$ be the subsets of Σ that actually appear in $R1 \ldots Rp$ respectively. Let I be a set of time intervals, which may include semi-infinite intervals extending from some finite instant to inifinity. Each element in I is labeled by an element in Σ. Define T(I) to be the trace which has an element for each element in I and has the obvious partial order defined between elements whose time intervals are non-overlapping. Referring to Fig. 4, let

- **Ext** = set of time intervals labeled 'external',
- **Int** = set of time intervals labeled 'internal',
- **Seq**(j) = set of time intervals labeled 'sequencer' for sequencer SEQ_j.

For every interval in **Int** with label e there are corresponding intervals with the same label in **Ext** and in every **Seq**(j) such that $e \in \Sigma_{Rj}$, namely those which start at the same time. We assume that the starting points of intervals in **Int** lie within some finite time period of interest, and the intervals in **Ext** and **Seq**(j) are restricted to intervals corresponding to those in **Int.**

With this notation in place we state some propositions, or axioms, that describe the properties of the circuit of Fig. 3. These properties will be

used to prove that the circuit is safe and live. The propositions should be viewed as specifications for the correct behavior of lower level circuit modules and the external world, and will be justified in later sections.

Proposition 5. (External world protocol): *For all events e,*
1. REQ_e *is raised only if* ACK_e *is low.*
2. REQ_e *is lowered only if* ACK_e *is high.* □

Proposition 6. (Arbiter safety and liveness):
1. *For any events e1, e2 that are mutually exclusive,* ACK_{e1} and ACK_{e2} *are never high simultaneously.*
2. *For any event e,* ACK_e *is raised only if* IN_e *is raised.*
3. *For any event e,* ACK_e *is lowered only if* IN_e *is low, and within a finite time of* IN_e *being lowered.*
4. *Consider any set of events* $\Sigma' \subseteq \Sigma$, *such that no two events in* Σ' *are in the same path expression.*

Then if all IN_e, $e \in \Sigma'$, *are raised, within a finite time all* ACK_e, $e \in \Sigma'$, *must be raised.* □

Proposition 7. (Sequencer protocol): *For any sequencer* SEQ_j,
1. TA_e *is raised only if* TR_e *is high.*
2. TA_e *is lowered only if* TR_e *is low.*
3. DIS_e *is stable while all* TR*'s and* TA*'s are low.* □

Proposition 8. (Sequencer safety and liveness): *For any sequencer* SEQ_j, *assume that at all times,*
- *no two* TR*'s are high simultaneously,*
- TR_e *is raised only if* DIS_e *and all* TA*'s are low,*
- TR_e *is lowered only if* TA_e *is high.*

Then the following hold:
1. TA_e *is raised within a finite time of* TR_e *being raised.*
2. TA_e *is lowered within a finite time of* TR_e *being lowered.*
3. *For any sequencer* SEQ_j, *whenever all* TA*'s and* TR*'s are low, exactly those events e will have* DIS_e *low, for which* $S(T(\textbf{Seq}(j)))$ *can be extended by e to give a prefix of some sequence in* L_{Rj}. □

Proposition 9. (Initialization)
1. *Sequencers are initialized with all* TA*'s low.*
2. *The synchronizer circuit SR flip-flops are initialized to make all* CLR*'s high.* □

The following theorem states that a synchronizer satisfying Propositions 5 through 9 is provably safe.

Theorem 10. (Synchronizer Safety): $T(\textbf{Ext}) \in Tr_\Sigma(M)$.

Proof. See the appendix. □

As a converse to Theorem 10 we would like to show that our circuit can produce any valid trace **Ext**, such that $T(\textbf{Ext}) \in Tr_\Sigma(M)$ for at least some behavior of the external world. However for some traces $T \in Tr_\Sigma(M)$, there does not exist any **Ext** such that $T(\textbf{Ext}) = T$, so there is no way any circuit can produce the required trace **Ext**. This happens when T does not sufficiently constrain the order in which the elements may occur so that any actual set of time intervals will have fewer concurrent elements than T. Given such a T it is necessary to constrain its partial order relation further, by adding additional (consistent) precedence relationships. It is easy to show using Definition 4 that this will never remove T from the set $Tr_\Sigma(M)$. We shall show that whenever T is sufficiently constrained so that it falls in a class of traces we call *layered,* then for some behavior of the external world $T(\textbf{Ext})$ for our circuit will equal this modified T.

Definition 11. A trace $P = (Q, \leq, L)$ is called *layered,* if Q can be subdivided into a sequence of *subsets,* such that for any *i1, i2* $\in$ Q, *i1* precedes *i2* iff the *subset* in which *i1* lies precedes the *subset* in which *i2* lies. □

The trace in Fig. 1 is layered, since its elements can be subdivided into the sequence of *subsets* $\{(A_1),(B_1,C_1),(A_2),(B_2,C_2),(A_3),(B_3,C_3)\}$ with the above property. If the size of each *subset* were one, then the trace would be totally ordered.

In general, any trace P will have a corresponding layered trace T which preserves most of the parallelism of P. It is easy to show that for any trace P, there exists a layered trace T, which differs from P only in that the partial order relation of P is a restriction of that of T.

Theorem 12. (Synchronizer Liveness): *Given any layered trace* $P \in Tr_\Sigma(M)$, *our circuit will produce an event trace* **Ext**, *such that* $T(\textbf{Ext}) = P$ *for some behavior of the external world.* □

Proof. See the appendix. □

4 Implementing the sequencer for a simple path expression

This section shows how to construct a sequencer that enforces the semantics of a simple path expression. The sequencer circuit is constructed in a syntax-directed fashion based upon the structure of the simple path expression. We show that a com-

pact layout for the sequencer exists, so that circuits of this type can be implemented economically in VLSI.

Since a simple path expression is a regular expression, the sequencer for a simple path expression is similar to a recognizer for the regular expression. Although schemes for recognition of regular languages have been proposed that avoid broadcast [4], we will use a scheme that requires broadcast of events throughout the sequencer [5, 12]. Because our scheme for interconnecting sequencers (see Sect. 3) requires broadcast, the broadcast within an individual sequencer carries no additional penalty. A sequencer for a simple path expression is built up from primitive cells, each corresponding to one character in the path. The syntax of the path determines the interconnection of the cells in the sequencer. In this section, we first describe the behavior of a sequencer for a simple path expression, then give a syntax-directed construction method.

As noted in Sect. 3, a synchronizer communicates with each of its sequencers using three lines for each event:

- TR_e: a signal to the sequencer that event e is about to commence in the outside world;
- TA_e: an acknowledgement from the sequencer that the execution of event e has been noted by the sequencer.
- DIS_e: a status line indicating that action e would violate the path constraints so that TR_e should not be asserted by the outside world. It is valid when TR and TA are both low.

These communication lines interact in a complex way. For a single type of event, the signals TR_e and TA_e follow the four-cycle signaling convention (for an example see Sect. 3). For different types of events, the outside world must guarantee the correct interaction of TR signals by ensuring that only one TR signal for an event satisfying the simple path expression is asserted at any time. The outside world can use the DIS status lines to determine which requests to send to the sequencer.

The sequencer also has a part to play in ensuring the correct interaction of TR, TA and DIS. Besides generating a TA signal that follows the four cycle convention with TR, it must ensure that the signal DIS_e is correct as long as no TR or TA signal is asserted. This guarantee means that if neither TA nor TR is asserted, and neither DIS_{e1} nor DIS_{e2} is true, then the outside world may choose arbitrarily between *e1* and *e2*, letting either of them through to the simple path sequencer. On receiving a TR_e signal, then, the sequencer must assert TA_e, adjust its internal state to reflect the occurrence of event

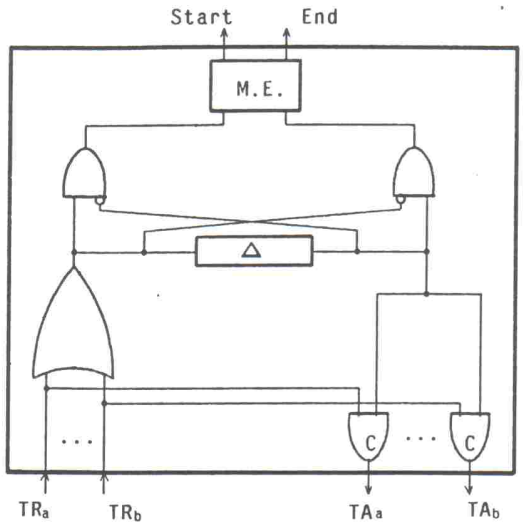


Fig. 5. The controller for path P

e, assert the proper set of DIS lines while awaiting the negation of TR_e before negating TA_e.

Now that the behavior of a sequencer has been described, we show how to construct a sequencer for any simple path expression. A sequencer has two parts: a controller and a recognizer. The controller is connected directly to the rest of the outside world and generates both the TA signals and some control signals for the recognizer. The recognizer keeps track of which events in the path have been seen and generates the DIS signals.

Figure 5 shows the controller for a simple path P. The controller accepts the signals TR_e from the synchronizer for each event e that appears in P. It generates the signals TA_e along with **Start** and **End**. The meaning of TA_e is that all actions caused by TR_e have been completed. In this realization, TA is just a delayed version of TR, where the delay is long enough to let the sequencer stabilize. An upper bound on this delay can be computed from the layout of the rest of the circuit. Thus the sequencer is self-timed but not delay insensitive. A more complicated, delay insensitive circuit will be described in a separate paper [1]. **Start** and **End** are essentially two phase clock signals that control the movement of data through the recognizer for P. Roughly **Start** is true from the time one TR is asserted until the corresponding TA is asserted, while **End** is true from the time TR is deasserted until TA is also deasserted. The element labeled M.E. (Mutual Exclusion) is an interlock element as shown in Fig. 12. It is required to guarantee that the two clock phases are strictly non-overlapping.

The recognizer for a path accepts the TR_e sig-

nals and generates the DIS signals. It is made up of sub-circuits corresponding to subexpressions of the path. To construct the recognizer for a path, we parse the path using a context-free grammar. Productions that are used in parsing the path determine the interconnections of sub-circuits to form the recognizer. Non-terminals that are introduced in the parse correspond to primitive cells used in the circuit.

Recognizers are constructed using the following grammar for simple path expressions.

S⟶ **path** R **end**

R⟶ R;R|(R+R)|(R)*|⟨event⟩.

The terminal symbols in the grammar correspond to primitive cells; there is one type of cell for the "+" symbol, one for the "*" symbol, one for the ";" symbol, and one for each event. The non-terminals correspond to more complex circuits that are formed by interconnecting the primitive cells. Using the method described in [3], semantic rules attached to the productions of the grammar specify how the circuits on the right of each production are interconnected to form the circuit on the left.

To keep track of which events in the path have occurred and which are legal, the sub-circuits of a recognizer communicate using the signals ENB (enable) and RES (result). If ENB is asserted at the input of a circuit for a subexpression at the beginning of a cycle (when START is asserted), the subcircuit begins keeping track of events starting with that cycle, and asserts RES after a cycle if the event sequence so far is legal for the subexpression. The ENB input may be asserted before any cycle, and the subcircuit must generate a RES signal whenever any of the previous ENB inputs by itself would have required it. At the top level ENB is asserted only once, before the first cycle. Between cycles each subcircuit deasserts the DIS signal for an event, if the occurance of that event during the next cycle is legal (this is the case if the subcircuit would assert RES for some subsequent sequence of events even if ENB were not asserted any more). These event signals from all subcircuits are combined to generate the external DIS signals.

Figure 6 shows the cell for event *e*. Two latches, clocked by **Start** and **End**, control the flow of ENB and RES signals. The latches are transparent when their enable is asserted and hold their previous value otherwise. The latch pair forms a level triggered master – slave D-Flip-Flop, clocked by the non-overlapping clock signals **Start** and **End**.

The event cell in Fig. 6 propagates a 1 from ENB to RES only if event *e* occurs. When this cell is used in a recognizer for a path expression, the

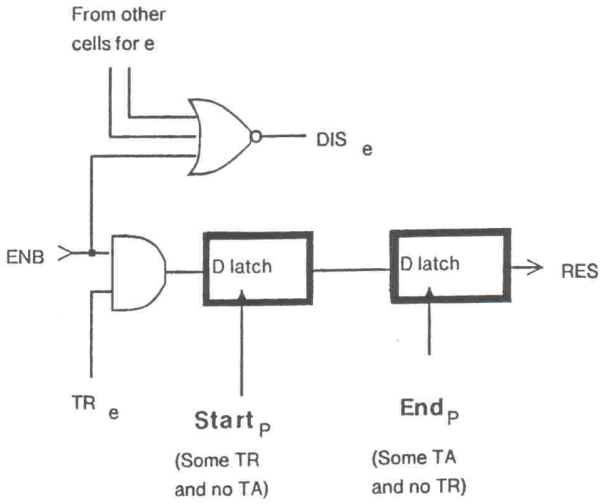


Fig. 6. Cell for event *e* in path P

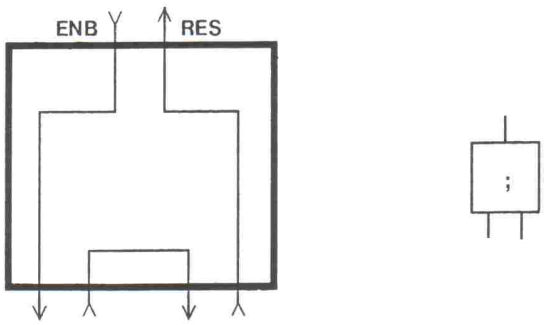


Fig. 7. Cell for ";"

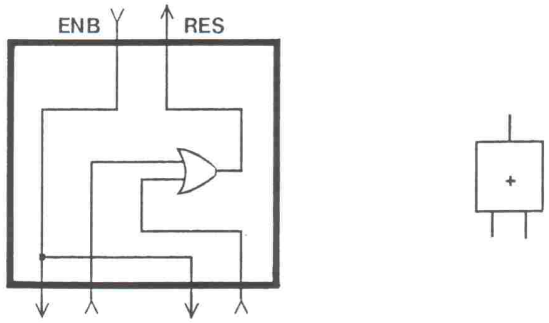


Fig. 8. Cell for "+"

ENB input will be true if and only if event *e* is permitted by the expression. Thus, if ENB is true it negates DIS_e for the path, as shown in the figure. When a request TR is made, the output of the AND gate is loaded into the leftmost latch. If this request is TR_e, this output is 1; otherwise it is 0. In either case the output of the AND gate is propagated to RES through the latch when TR is lowered.

Figures 7 and 8 show the cells for the ";", (sequencing) and "+" (union) operators. These are strictly combinational circuits. The circuit for ";" feeds the RES signal from the circuit at its left into

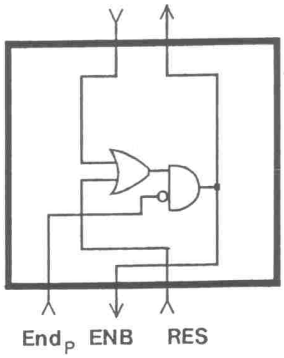


Fig. 9. Cell for "*"

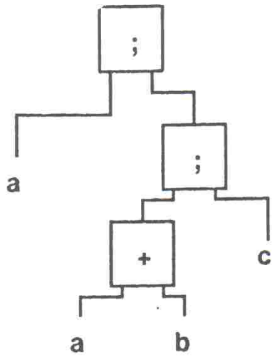


Fig. 10. A recognizer for **path** a;(a+b);c **end**

the ENB signal for the circuit to its right. The circuit for "+" broadcasts its ENB signal to its operands and combines the RES signals from its operands in an OR gate.

Figure 9 shows the cell for the "*" operator. The cell enables its operand after receiving either a 1 on either its own ENB or its operand's RES. Every time the operand is enabled the "*" cell also puts out a 1 on its own RES. It therefore outputs 1 on RES after 0 or more repetitions of its operand's expression. The additional AND gate sets the output to 0 momentarily after each event, thereby preventing the formation of a latch when two or more "*" cells are used together. This cell is responsible for making the minimum cycle duration depend on the path expression. During the first phase of a cycle the sequencer has to perform an ε-closure of the simple path expression. This delay is directly reflected in the gate delay between the ENB input and RES output of the "*" cell. These delays will add up for an expression like ((a*;b*);(c*;d*)).

When larger circuits are made from these cells, the RES and ENB signals retain their meanings. Each event cell or sub-circuit formed from several cells accepts one input ENB and produces one output RES. In general we define a pair of ENB and RES to be correct if the following applies at the beginning of each cycle (just before START is asserted):

- ENB is true if and only if the sequence of events so far can be extended by any sequence of events satisfying the regular expression of the subcircuit controlled by the ENB/RES pair, to give a prefix of some sequence in L_{Rj}.
- RES is true if and only if some sequence of events satisfying the subexpression has just completed, and ENB was true just before the beginning of that sequence.

In addition, a sequencer has a signal INIT, not shown in the figures, which clears the RES outputs of all event (leaf) cells and generates the ENB input for the root cell (which must a "*" cell, if there is an outermost implied Kleene Star) during the first cycle (an RS flip-flop set by the INIT signal and reset by END can be used to generate this ENB signal).

Figure 10 shows a recognizer for the path **path** a;(a+b);c **end** constructed using this syntax-directed technique.

All recognizers constructed by the previously described procedure perform the correct function, as required by Propositions 7 and 8. The former follows directly from the control circuit while the latter is equivalent to the following: If a recognizer is initialized and some sequence of events 'clocked' into the circuit, the recognizer will output 1 on DIS_e between cycles for precisely those events e that are forbidden (as the next event) by the simple path expression. To prove this we show that the ENB input of an event cell in the recognizer is 1 if and only if the event corresponding to this cell is permitted by the path. As shown in Fig. 6, DIS_e is 1 if and only if none of the cells for event e is enabled. Therefore, proving that an event cell has its ENB signal set if and only if the corresponding event is permitted in the path will show that the recognizer is functionally correct. In other words, we wish to prove that all ENB signals for event cells are correct, according to the definition of ENB above.

We outline a proof of the stronger statement that all ENB signals in the recognizer are correct. This proof is based upon the structure of the recognizer. An ENB signal in a recognizer is set by one of four sources:

- The operand port of a "+" or "*" cell;
- The left operand port of a ";" cell;
- The right operand port of a ";" cell;
- The INIT signal.

In the first and second cases the signal is correct if and only if ENB for the operator cell is correct.

In the third case the signal comes from the RES port of a recognizer for an initial subexpression. Therefore it is correct if and only if the RES signal for the subexpression is correct. In the fourth case the signal is asserted only at the start of the recognition and is correct by definition. Thus, to prove that the circuits are correct, we need only prove that if the ENB signal for any recognizer is correct then so is the RES signal.

Once again, the proof of correctness is based upon the structure of a recognizer. In a correct recognizer the RES signal is true at time t_1 if and only if the ENB signal is true at some preceding time t_0 and the events between t_0 and t_1 obey the path. A recognizer that is a single event cell is clearly correct. A recognizer for path **a;b** built by composition of correct subrecognizers for **a** and **b** is also correct, since if RES_b is true at time t_2 then there must be some time t_1 when RES_a was true, with all intervening events satisfying path **b**. But then there must have been a time t_0 when ENB_a was true and all events between t_0 and t_1 must satisfy path **a**. By definition of composition, then, the events between t_0 and t_2 satisfy **a;b**. A recognizer for path (**a**)* is correct if its subrecognizer is correct, since it outputs 1 and enables its operand if and only if ENB or RES_a is true. Finally, a recognizer for path **a**+**b** is correct if both subrecognizers are correct, since if RES is true then one of RES_a or RES_b must be true, and if one of ENB_a or ENB_b is true then ENB must be true. Since all methods of constructing recognizers have been shown to lead to correct circuits, recognizers, constructed using this procedure are functionally correct.

Finally, we give a compact floor plan for the circuit. The floor plan for a sequencer, shown in Fig. 11 has the cells that make up the recognizer arranged in a line with the controller to one side. The TR signals flow parallel to the line of recognizer cells to enter the controller, and the **Start** and **End** signals emerge from the controller to flow parallel to the line of cells. The ENB and RES signals that are used for intercell communication also flow parallel to the line of cells.

The layout in Fig. 11 is fairly small. If the sequencer for a path of length n that has k types of input events is laid out in this fashion, the area of the layout is no more than $O((n+k)(\log n+k))$. This is due to the structure of the recognizer circuits. All recognizer circuits are trees, which can be laid out with all nodes on a line and edges running parallel to the line using no more than $O(\log n)$ wiring tracks [8]. Thus the height of the circuit in Fig. 11 is $O(\log n+k)$ while its width is $O(n+k)$.

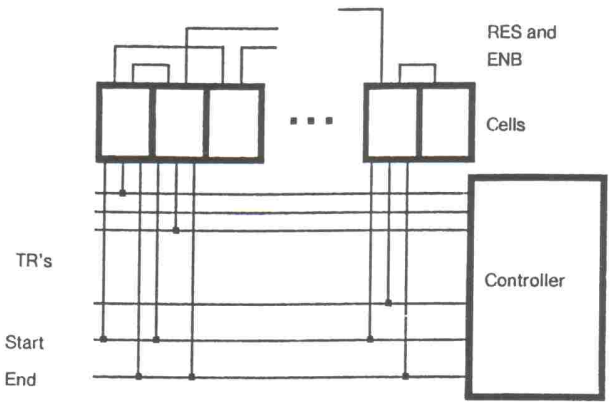


Fig. 11. The floor plan for a sequencer

5 Implementation of the arbiter

In this section we elaborate on the arbiter shown in Fig. 3 to show that the conditions assumed for it can be met. In older literature the term arbiter refers to a device that selects a single event from a mutually exclusive set of requests. In this paper the term is used in a somewhat less restrictive sense. All events need not be mutually exclusive and the arbiter may select more than one event concurrently, as long as the mutual exclusion conditions are satisfied. We first show how such an arbiter can be built. Later we discuss ways to ensure that the arbiter is *fair* when forced to chose between events. This is much harder to achieve than just the mutual exclusion requirement.

We shall make use of the following terms: An event is *pending* from the time the external world asserts the request until the time the circuit asserts the acknowledge for the event. An event is *enabled* if it is pending, and it is not currently disabled by any path.

The following observation helps to simplify the arbiter: a pair of events occurring in any single path expression must be mutually exclusive. This is due to the role that each event plays in enforcing synchronization among a set of multiple path expressions, all containing the same named event. The arbitration function can thus be represented by a *conflict graph*, in which each event is denoted by a vertex and the relation between a pair of mutually exclusive events is denoted by an undirected edge. From our observation, it follows that the resulting conflict graph for a set of path expressions consists of a set of overlapping cliques, where a clique of k nodes, $A_1, A_2, \ldots, A_k$, corresponds to a simple path expression R, with $\Sigma_R = \{A_1, A_2, \ldots, A_k\}$. The conflict graph represents the static structure of a multiple path expression. Fig-

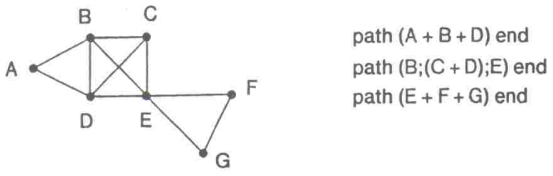


Fig. 12. The conflict graph of a path expression

ure 12 shows a multiple path expression and its conflict graph.

The dynamic behavior of the arbiter depends on the conflict graph together with the set of events that are *enabled* at any instant. The dynamic structure of a multiple path expression is represented by an *active* subgraph of the conflict graph induced by the set of vertices corresponding to the events enabled at that instant. The function of the arbiter is to select an independent set of this subgraph, thus ensuring that only one of any pair of mutually exclusive events is enabled. In this paper we require the arbiter to respond whenever it can and not introduce deliberate wait states. More formally we define a *maximally parallel set* of events to be an independent set of the active subgraph, such that it is not a subset of any other independent set of the active subgraph. We require the arbiter to respond with a maximally parallel set without introducing deliberate delays. In general there will be more than one possible maximally parallel set, and the arbiter need not choose the largest one.

Before proceeding further, let us consider the path expression **path** A + B **end**, where the conflict graph is $G=(V, E)=(\{A, B\}, \{[A, B]\})$. Seitz [16] has shown how to build an arbiter for such a structure using an interlock-element, as shown in Fig. 13.

Circuit operation in Fig. 13 is most easily visualized starting with neither client requesting, v_1 and v_2 both near 0 volts, and both outputs high. If any single input, say A_{in}, is lowered then v_1 is driven high, resulting in A_{out} being lowered – B_{out} remains unaffected. Moreover, once A_{out} is lowered, and as long as A_{in} is kept low, the interlock element remains in this stable state irrespective of what happens to B_{in}. If A_{in} is now raised high, then the element returns to its initial condition, if B_{in} is still high; or B_{out} is lowered, if B_{in} is lowered in the meantime.

However, the interesting situation occurs when both A_{in} and B_{in} are both lowered concurrently, i.e., within a very short interval of time. In this case the cross-coupled NOR gates enter a metastable state, which is resolved after indeterminate period of time in favor of either A or B. Since this resolution depends on the thermal noise generated by

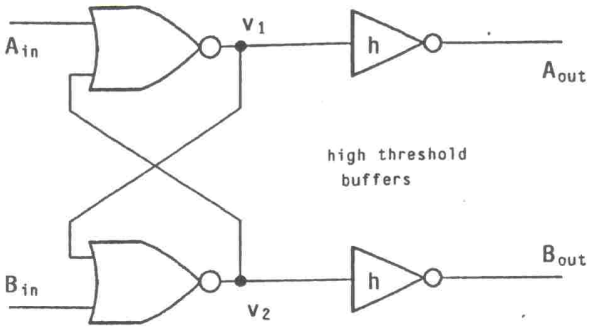


Fig. 13. Seitz's Interlock Element

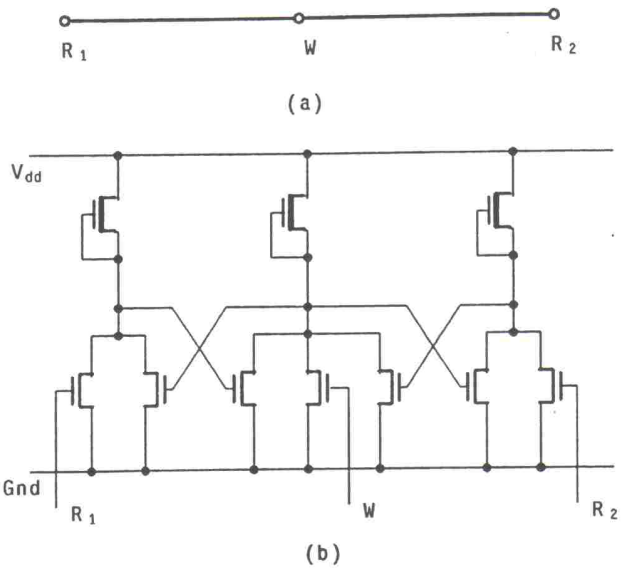


Fig. 14. a The Conflict Graph and **b** The Arbiter in NMOS

the gates, it is inherently probabilistic. In this case the outputs of the NOR gates themselves cannot be used as the outputs. High threshold inverters between the NOR gates and the outputs prevent false outputs during the metastable condition.

Seitz's idea can be extended by generalizing it to the conflict graph for an arbitrary set of path expressions. Roughly speaking, we may transform the conflict graph to a circuit by replacing each vertex with a NOR gate and each edge with a cross-coupling of NOR gates corresponding to the pair of vertices on which the edge is incident. Consider the circuit for the readers-writers path expression:

path $R_1 + W$ **end**
path $R_2 + W$ **end**

where the pair R_1 and W and the pair R_2 and W are mutually exclusive. The conflict graph and the circuit for this expression are shown in Fig. 14.

The above arbiter is quite satisfactory if we do not require the arbiter to be fair. In certain applications, however, it is required that the arbiter should be fair when faced with a choice. So far we have not defined what we mean by fairness. The most commonly used definitions of fairness that allow pending events to be disabled are due to Lehman, Pneuli and Stavi [9]. The definitions apply to total extensions of infinite exccution traces. An arbiter is fair if every total extension of any infinite execution trace, is fair. Only events that are continuously requesting ('continuously pending') are considered.

1. **Impartiality.** Each event is *infinitely often* acknowledged. (Must be fair to all events).
2. **Fairness.** Each event is either *infinitely often* acknowledged or *almost everywhere* disabled. (Need be fair only to events that are *infinitely often* enabled).
3. **Justice.** Each event is either *infinitely often* acknowledged or *infinitely often* disabled. (Need only be fair to events that are *continuously* enabled).

The order of these definitions is such that if an arbiter is fair according to one definition it will also be fair according to any succeeding definition, but not the converse. Note that these definitions do not require different events to be acknowledged with equal promptness, all that is required is that no event is starved.

Let us digress for a moment, a see why the arbiter described earlier fails to enforce any of these forms of fairness. The arbiter implementation based on the extension of Seitz's interlock element does make arbitrary choices at times, but such an implementation in NMOS has some problems, even if it is assumed that all equal sized transitors are perfectly balanced. Consider the circuit for the readers-writer problem illustrated in Fig. 14. Consider the situation when the circuit is in the none-requesting condition and all three requests, R_1, R_2 and W, arrive concurrently. An infinitesimally short interval Δt after all three requests arrive, let us assume that the voltages at the outputs (of the NOR gates) have increased by an infinitesimally small value $\Delta v \ll v_{th}$. The pull-down MOS transistors may be assumed to be operating in their linear region. If all pull-ups are assumed to provide equal active resistance, the output of the NOR gate corresponding to W will grow less rapidly than those corresponding to R_1 or R_2. The cumulative effect of this imbalance will result in a low output for W's NOR gate and high outputs for R_1's and R_2's. Hence if R_1, R_2 and W request continuously then the request for W will never go through, resulting in W's starvation. It is easy to see that this violates all three definitions of fairness.

Since we do not allow deliberate wait states it is not possible for an arbiter for path expressions to be fair according to the first definition. Consider for instance the following path expression:

path (A + B); C **end,**
path D; (A + E) **end**

Suppose that each event takes the same amount of time to execute externally and that new requests for each event are forthcoming as soon as allowed by the protocol. Then simultaneous execution of D and B will alternate with simultaneous execution of C and E without the arbiter ever having to block any event. Yet, event A will never execute even if it remains continually ready. If, however, the first request for event B is delayed by the time it takes to execute an event, then initial execution of event D may be followed by alternate executions of A and (D,C)! Note that neither the duration of external events nor the occurrence of external requests is under the control of the circuit.

The second (and therefore third) definition of fairness can be enforced using a simple LRU type deterministic arbitration algorithm. Assume there are k events. We assign a priority number from 0 to k − 1 to each event, where the priority corresponds to the number of times the event is *blocked*, i.e., the number of times the event is enabled but not selected by the arbiter. At any instant the arbiter selects from the set of enabled events in order of priority. When an enabled event is selected its priority number is reinitialized to the lowest value. On the other hand, if the enabled event is not selected its priority number is incremented by one. Since each event must be enabled an infinite number of times, since any particular event can have at most k − 1 neighbors in the conflict graph, and since each time it is blocked at least one of its neighbors is selected with a resulting increment in its own priority, after the k^{th} attempt it will have a priority of k − 1, the highest possible. It is possible to show (using induction on k) that no more than one event can ever get a priority of k − 1. Hence when the event gets enabled next it will have the highest priority and get selected. Since this will happen an infinite number of times, this ensures fairness according to the second definition. The LRU algorithm has the added advantage that the response time to different events is approximately balanced.

The following simple circuit can be used to im-

plement the LRU algorithm. It uses the arbiter circuit described previously to enforce the mutual exclusion. However each input is preceded by k switchable delay lines (k^2 delay lines in all). Each delay line can be switched off digitally, for instance by selectively bypassing the delay line. The delays are chosen large enough, and their variation made small enough, so that if one input is delayed by fewer delay lines than another, it will be selected by the arbiter (because the arbiter didn't notice the other one in time). The delay lines for any event are controlled by priority of the event: each time an event is blocked, an additional delay line is switched off for it, whereas if the event is acknowledged all its delay lines are switched on again, reducing its priority to the lowest level. A complete arbiter circuit based on this idea requires just $O(k^2)$ area.

The second definition is not the strongest possible form of fairness that can be enforced for path expressions. Consider for instance the path expression **path**((A;C)+(B;A))**end**. As before assume that all events are pending at all times. The execution sequence BABABA ... is fair according to this definition even though event C is starved (event C is never enabled). We could have done better, however, since ACBAACBA ... is also a legal execution sequence.

Although we do not know the strongest form of fairness enforcible for path expressions, it obviously lies somewhere between Definitions 1 and 2. Intuitively, the fairest arbiter would only cause starvation for the least number of events possible. The problem can be greatly simplified by requiring the arbiter to be *oblivious* of the sequencing constraints and therefore equate a disabled event with an event not requesting. This restriction will also tend to simplify the logic since the arbiter size need not depend on the size of the path expressions, but only on the alphabet size. It should be kept in mind however that like our previous restriction requiring prompt response, this restriction limits the kind of arbiters possible. It may be noted that the LRU arbiter described previously is oblivious.

We shall describe a probabilistic arbitration algorithm for an oblivious arbiter whose infinite execution traces will be "fair" with probability 1 where "fair" is defined by either of Definitions 2 and 3. It also holds for stronger forms of fairness and therefore realizes a type of fairness between Definitions 1 and 2. The algorithm is as follows: Whenever the set of currently executing events is not a maximally parallel set, find all ways of extending this set with enabled events so that the new sets are maximally parallel, choose one of them at *random*, and then acknowledge the events in the selected extension. Every time an event is no longer disabled there is a non-zero probability that it will be acknowledged, and if this is the case infinitely often the event will be infinitely often acknowledged. It follows that this algorithm ensures fairness in the sense of the second or third definition above. It will also prevent starvation for event C in the last example above.

Although we do not know of any direct implementation we shall describe a way of implementing this arbitration algorithm using an oracle for generating random bits. The oracle can be practically realized in a separate isolated circuit that uses amplified thermal noise to generate a random bit pattern. Again we use the extension of Seitz's interlock element, described previously, as the starting point and add a delay element at each input. The delay elements can be digitally switched on or off (by bypassing them), and are large enough so that if two conflicting events are enabled at the same time, and one is delayed by the delay element, the other is sure to be passed by the arbiter. This means that the delay should exceed the gate delay of the arbiter (when no conflicts occur). The delay elements are each controlled by a 1 bit register, which determines if the delay is on or off. A new value is loaded into each register from a (separate) oracle each time the corresponding event gets enabled. This means whenever a new set of events gets enabled, their 'priorities' are randomly 1 or 0. It is easy to show that any maximally parallel set then has a nonzero probability of being selected (when just its events have priority 1 and all others have priority 0), which is just what the probabilistic algorithm requires. To ensure that the random bits clocked into the different registers are largely uncorrelated, the oracle is split into multiple oracles by clocking it into a shift register at a high rate. A tapped delay line could be used instead of the shift register.

Finally, we show that no deterministic oblivious arbiter can do as well as our probabilistic arbiter. We show that every deterministic oblivious arbiter gives rise to starvation of an event which is continually requesting for some path-expression for which the probabilistic algorithm (described above) does not cause such starvation.

The difficulty of building a fair deterministic arbiter that matches the probabilistic arbiter can be illustrated by an example. Consider the following path expression:

path (A;C)+(B;(A+B)) **end.**

Assume the LRU algorithm, described previously, is being used, and that the external clients always request permission to perform all three events A, B and C. Let the priorities of all three be 0's initially. As a result, initially A and B are enabled. Assume that B is selected, making B's priority 0 and A's priority 1. In the next instant, A and B will again be enabled. But now A has the higher priority and will be selected, so that A's priority becomes 0 and B's becomes 1. Continuing in this fashion, it is easy to see that the sequence chosen will be BABABA.... The trouble with this scheme is that C will never be enabled even if its request is pending. Increasing the number of levels of priority will not help. This example can be extended to the following Lemma.

Lemma 13. *Let M be a deterministic finite-state transducer implementing an oblivious deterministic arbiter. Then there exists a path expression over $\Sigma=\{A,B,C\}$ such that one event, say C, will be starved even though its request is continually pending. Moreover the probabilistic algorithm does not cause such starvation for this path expression.*

Proof. Let M be a deterministic finite-state transducer whose alphabet is $\Sigma=\{A,B,C\}$. Let the states of M be $S=\{s_1,s_2,\ldots,s_m\}$. Let the conflict graph, G, for the path expression to be complete graph on the vertices A, B and C. We construct a path expression P with the conflict graph G such that M causes the starvation of the event C. Notice that because of the nature of the conflict graph G, if at any instant A and B (but not C) are enabled then at most one of A and B may be selected by M.

Let s_1 be an arbitrarily chosen state of M. We conduct an experiment on M by continuously providing A and B as the enabled inputs, starting with M in the state s_1. If we present a string of inputs $\{A,B\}, \{A,B\}, \ldots, \{A,B\}$ of length m then we notice that at the 1st input ($\{A,B\}$, the transducer deterministically goes from the state $s(1)=s_1$ to a state $s(2)$ while outputting A or B. Let $s(1)$, $s(2), \ldots, s(m+1)$ be the sequence of states and $\sigma\in\{A,B\}^m$ be the output string produced as a result of the experiment. As a consequence of the pigeon-hole principle, some two states in the sequence of states will be the same. Of all such pairs, let $s(i)$ and $s(j)$ be two such states closest to s_1. Assume that $i<j$ and k the smallest multiple of $(j-i)$ such that $k\geq i$. Without loss of generality assume that M outputs B when in state $s(i)$ with the input $\{A,B\}$.

Let P be the path expression

path $(A+B)^{i-1};((A;C)+B);(A+B)^{k-i}$ **end**

It is easy to see that P has G as the conflict graph and if the requests for A, B and C are continuously pending then the sequence of outputs will be a string in $\{A,B\}^\omega$ and C will never be enabled.

The probabilistic algorithm would have no problem with the path-expression since from any state (of the path expression) it could reach the state enabling C with non-zero probability, and hence enable C an infinite number of times in an infinite trace. □

The result of the above lemma can also be stated as follows: A deterministic oblivious arbiter needs at least N/2 states to do as well as one using the probabilistic algorithm, where N is the size of the path-expression, whereas the probabilistic algorithm requires a constant number of internal states. The actual bound on the minimum number of states required may be much larger.

However, for many path expressions the LRU algorithm is just as fair as the probabilistic algorithm and has the advantage that the response times are approximately balanced, instead of being a complex function of the conflict graph as in the probabilistic algorithm. For such path expressions the use of the LRU algorithm is preferable. The problem of determining just which path expressions satisfy this property, and well as more direct ways of combining the advantages of the LRU algorithm with those of the probabilistic algorithm remain to be investigated.

6 Conclusion

Since our circuits have the constant separator property, a more compact O(N) layout is possible using the techniques of [5]. However, while it is definitely possible to automatically generate the $O(N\cdot\log(N))$ layout that we propose, it is much more difficult in practice to generate the O(N) layout of [5]. Furthermore, the O(N) layout will occupy less area only for very large N. We suspect that ease of generating the layout will win over asymptotic compactness in this case. One of the authors (M. Foster) is currently implementing a silicon compiler for path expressions, based on the ideas in this paper.

Finally, we plan to investigate extensions of our construction to appropriate finite state subsets of CSP [6] and CCS [11]. In the case of CSP the subset will only permit boolean valued variables

and messages which are signals. If the number of message types is fixed, we conjecture that area bounds comparable to those in Sect. 4 can be obtained. Arrays of processes in which the connectivity of the communication graph is low can be treated specially for a more compact layout. Such a finite-state subset of CSP may even be more useful than the path expression language discussed in the paper for high level description of various asynchronous circuits.

Acknowledgements. The authors wish to acknowledge the help of K. Karplus. His comments on an early draft of this paper contributed significantly towards improving the clarity of the paper.

References

1. Anantharaman TA (1985) A delay insensitive regular expression recognizer
2. Campbell RH, Habermann AN (1974) The specification of process synchronization by path expressions. In: Goos G, Hartmanis J (eds) Lecture notes in computer science, vol 16. Springer, pp 89–102
3. Foster MJ (1984) Specialized silicon compilers for language recognition. PhD Th, CMU July 1984
4. Foster MJ, Kung HT (1982) Recognize regular languages with programmable building-blocks. J Digital Syst VI(4):323–332
5. Floyd RW, Ullman JD (1982) The compilation of regular expressions into integrated circuits. J Assoc Comput Mach 29(3):603–622
6. Hoare CAR (1978) Communicating sequential processes. Commun ACM 21(8):666–677
7. Lauer PE, Campbell RH (1974) Formal semantics of a class of high-level primitives for coordinating concurrent processes. Acta Inf 5:297–332
8. Leiserson CE (1981) Area-efficient VLSI computation. PhD Th, Carnegie-Mellon University
9. Lehman D, Pnueli A, Stavi J (1981) Impartiality, justice and fairness: The ethics of concurrent termination. Automata, Languages and Programming, pp 265–277
10. Li W, Lauer PE (1984) A VLSI implementation of Cosy. ASM/121, Computing Laboratory, The University of Newcastle Upon Tyne, January, 1984
11. Milner R (1980) A calculus of communicating systems. Lecturd notes in computer science, vol 92. Springer, Berlin Heidelberg New York
12. Mukhopadhyay A (1979) Hardware Algorithms for nonnumeric computation. IEEE Trans Comput C-28(6):384–394
13. Patil S (1975) An asynchronous logic array. MAC TECHNICAL MEMORANDUM 62. Massachusetts Institute of Technology
14. Pratt VR (1982) On the composition of processes. Symposium on Principles of Programming Languages, ACM January, 1982
15. Rem M (1983) Partially ordered computations, with applications to VLSI design. Eindhoven University of Technology
16. Seitz CL (1980) Ideas about arbiters. LAMBDA First Quarter, pp 10–14

Appendix: Proof details

Refer to Sect. 3:

Lemma 14. *If the same assumptions as in Proposition 8 are satisfied, then* $T(\mathbf{Seq}(j))$ *is consistent with* R_j.

Proof. From Proposition 8 it follows that **Seq**(j) consists of non concurrent time intervals. The result is therefore easy to prove by induction on the number intervals in **Seq**(j), using the same proposition. □

Lemma 15. *For each element i in* **Int** *with label e, the corresponding elements in* **Ext** *and* **Seq**(j) *are subintervals of i.*

Proof. Follows from the properties of the circuit in Fig. 3 (see also Fig. 4). □

Lemma 16. *For any* $Rj \in M$, $T(\mathbf{Int})|_{\Sigma_{Rj}}$ *is a totally ordered multiset.*

Proof. It is easy to show that $T(\mathbf{Int})|_{\Sigma_{Rj}} = T(\mathbf{Int}|_{\Sigma_{Rj}})$. But $\mathbf{Int}|_{\Sigma_{Rj}}$ consists of 'internal events' of the path expression Rj, during each of which the corresponding ACK is high. Hence by Proposition 6, no two such events overlap, and therefore $T(\mathbf{Int})|_{\Sigma_{Rj}}$ is a totally ordered multiset. □

Lemma 17. *For any* $Rj \in M$, $T(\mathbf{Int})|_{\Sigma_{Rj}} = T(\mathbf{Ext})|_{\Sigma_{Rj}}$.

Proof. For any element *i* of $T(\mathbf{Int})$, that is also in $T(\mathbf{Int})|_{\Sigma_{Rj}}$, the corresponding element of $T(\mathbf{Ext})$ will be in $T(\mathbf{Ext})|_{\Sigma_{Rj}}$ (Definition 2) since they must map to the same alphabet $e \in \Sigma_{Rj}$. Hence these traces have the same number of elements. Also from Lemma 15 it follows that if *i1* and *i2* are two elements of $T(\mathbf{Int})|_{\Sigma_{Rj}}$ satisfying one or none of "*i1* precedes *i2*" and "*i2* precedes *i1*", the corresponding elements of $T(\mathbf{Ext})|_{\Sigma_{Rj}}$ will satisfy at least the same relationships. In other words the partial order of $T(\mathbf{Int})$ is a restriction of that of $T(\mathbf{Ext})$. But by Lemma 16 $T(\mathbf{Int})|_{\Sigma_{Rj}}$ is a totally ordered multiset. Hence from the above $T(\mathbf{Ext})|_{\Sigma_{Rj}}$ will have the same partial order relationship and, therefore, be the same totally ordered multiset. □

Lemma 18. *For any* $Rj \in M$, $T(\mathbf{Sem}(j)) = T(\mathbf{Int})|_{\Sigma_{Rj}}$.

Proof. Follows from Lemma 15 and 16 in the same way as in the Proof of Lemma 17. The only difference is that $T(\mathbf{Seq}(j))|_{\Sigma_{Rj}} = T(\mathbf{Seq}(j))$. □

Lemma 19. *For any sequencer* SEQ_j, *no two* TR*'s are high simultaneously.*

Proof. The two TR's would be two ACK's of events in the same path expression Rj, which cannot be high simultaneously by Proposition 6. □

Lemma 20. *For any sequencer* SEQ_j, TR_e *is raised only if* DIS_e *is low and all* TA*'s are low.*

Proof. By induction on the number of rising transitions of TR's:

1. (First transition): Let the corresponding event be *e*. By Proposition 9 initially all TA's are low, and all CLR's are high, hence all TR's are low initially. By Proposition 7 all TA's will remain low until the first rising transition of TR_e. By the same proposition DIS_e will not change until the first rising transition of TR_e. If DIS_e were not low, IN_e would remain low (see Fig. 3). Hence by proposition 6, TR_e would remain low, a contradiction.

2. (For a succeeding transition): Let the corresponding event by p and that of the previous transition q. While TR_q is high no TA or TR other than TA_q or TR_q can be high (Proposition 6 and Lemma 19). Until CLR_q goes high, TR_q must remain high (see Fig. 3). Once CLR_q goes high, all IN_a, with $a \in \Sigma_{Rj}$, will be low after a short delay (see Fig. 3). Assuming the variation in this delay for different a's is less than the delay of the arbiter in lowering TR_q, all TR_a with $a \neq q$ will continue to remain low until CLR_q is lowered (see Fig. 3). All TA_a, with $a \neq q$, also continue to remain low (Proposition 7). But CLR_q remains high at least until TA_q is lowered (see Fig. 7). Hence by the time TR_p is raised all TA's will be low. Also TR_p could not have been raised if IN_p were low (Proposition 6). But if DIS_p was high when TA_p was last lowered then IN_p would now be low (see Fig. 3), assuming the main NOR gate plus the 2-input NOR gate have a lesser delay than the Muller-C element plus the SR Flip-Flop. Moreover, DIS_p cannot change before TR_p is raised (Proposition 7). Hence DIS_p must be low when TR_p is raised. □

Lemma 21. *For any sequencer* SEQ_j, TR_e *is lowered only if* TA_e *is high.*

Proof. The NOR gate arrangement in front of the arbiter insures that once TR_e is high it remains high until CLR_e is raised, and this can occur only if TA_e is high (see Fig. 3). Moreover once TA_e is high it will remain high until TR_e is lowered (Proposition 7). □

Theorem 10.

Proof. Lemmas 19–21 satisfy the preconditions of Proposition 8. Hence T(**Seq**(j)) is consistent with Rj for any Rj∈M. By Lemma 18 and Definition 4, T(**Int**) is consistent with Rj for any Rj∈M. By Lemma 17 and Definition 4, T(**Ext**) is consistent with Rj for any Rj∈M. Hence we Definition 4, T(**Ext**) $\in Tr_\Sigma(M)$. □

Lemma 22. *If* $T \in Tr_\Sigma(M)$ *is layered, then each subset (cf. Definition 11) of* T *has the property that no two elements in it are instances of events in* Σ_{Rj} *for any* Rj∈M.

Proof. Any two elements *i1, i2* (corresponding to events *e1, e2*) in the same *subset* of T must be concurrent (Definitions 3, 11). Suppose *e1, e2* $\in \Sigma_{Rj}$ with Rj∈M. Then $T|_{\Sigma_{Rj}}$ will include *i1, i2*, which will be concurrent (Definition 2). Hence $T|_{\Sigma_{Rj}}$ cannot be a total order and therefore $T \in Tr_\Sigma(M)$ (Definition 4) – leading to a contradiction. Hence the result. □

Theorem 12.

Proof. The behavior we require of the external world is that it simultaneously raise REQ for all events in the first *subset* of T, wait until all corresponding ACK are high, then simultaneously lower all REQ, wait until all ACK are low, then repeat this *cycle* for the next *subset* of T, and so on. We need to show that under these conditions the circuit responds within a finite amount of time in each *cycle*. The result then follows directly.
As shown in the Proof of Lemma 20, all ACK's are initially low. Hence they are low at the beginning of each of the *cycles* mentioned in the previous paragraph. At the beginning of each such *cycle*, **Ext, Int** and every **Seq**(j) with Rj∈M, get redefined. Let Tp denote T restricted to subsets before the current cycle. It is easy to show by induction on the number of cycles and definition 4 that at the beginning of each cycle T(**Ext**) = Tp and $Tp \in Tr_\Sigma(M)$. Hence for any Rj∈M, $S(Tp|_{\Sigma_{Rj}})$ is a prefix of some element in L_{Rj}. If the next *subset* contains an instance *i1* of event *e1*, then for each Rj∈M such that $e1 \in \Sigma_{Rj}$, $S(Tp|_{\Sigma_{Rj}})$ can be extended by *i1* to give a prefix of some sequence in L_{Rj}; in fact this extension gives the next value of $Tp|_{\Sigma_{Rj}}$ (see Lemma 22). But by Lemmas 18, 17, for any Rj∈M, T(**Seq**(j)) = $T(\mathbf{Ext})|_{\Sigma_{Rj}} = Tp|_{\Sigma_{Rj}}$. Hence for each Rj∈M, such that e1 $\in \Sigma_{Rj}$, T(**Seq**(j)) can be extended by *i1* to give a prefix of some sequence in L_{Rj}. Thus by Proposition 8, the corresponding sequencers SEQ_j, with $e1 \in \Sigma_{Rj}$, will have DIS_j low. This applies to any *e1* in the next *subset* of T.
Therefore at the beginning of any cycle, when REQ_{e1} for any event *e1* in the next subset of T is raised, all DIS_{e1} inputs to the NOR gate for even *e1* (see Fig. 3), will be low. Also within a finite amount of time all relevant TA_{e1}'s must go low, by Proposition 8, since the corresponding TR_{e1}'s are already low. Hence CLR_{e1} will go low, and IN_{e1} will go high for each *e1* in the next subset of T. It follows from Proposition 6 and Lemma 22 that all ACK's corresponding to events in the next *subset* of T will be raised within a finite amount of time.
The Proof for the second half of the cycle is more straightforward. By Lemma 8 once all REQ's are lowered, within a finite time all relevant TA's will be raised, causing the corresponding CLR's to go high. As a result all relevant IN's go low (see Fig. 3) and hence by Proposition 6 all ACK's go low within a finite time, completing the cycle. □