NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

CMU-CS-85-173

On the Existence of Delay-Insensitive Fair Arbiters: Trace Theory and its Limitations

David L. Black
Carnegie-Mellon University
Pittsburgh, PA 15213

31 October 1985

Copyright © 1985 David L. Black

This research was partially supported by NSF Grant Number MCS-82-16706

Table of Contents

1. Introduction 1
    1.1. Background 1
    1.2. Contributions of this Paper 2
2. Trace Theory 2
    2.1. Introduction 3
    2.2. Delay Insensitivity 4
3. A Simple Example 5
    3.1. Udding's Non-Existence Result 6
4. Fairness 6
    4.1. Fairness and Arbiters 6
    4.2. Fairness and the Example 7
    4.3. A Fair Arbiter 8
    4.4. An Unfair Arbiter 11
5. Infinite Traces and Trace Theory 12
    5.1. Example 13
    5.2. Arbiter Automata 14
6. Composition Operator 15
    6.1. Notational Conventions 15
    6.2. Directed Traces 16
    6.3. Wire Trace Structures 16
    6.4. Operator Definition 18
    6.5. Composability 21
7. Liveness 24
    7.1. Example 24
    7.2. Changes to the Theory 25
8. Applications of the Composition Operator 27
    8.1. Delay Insensitivity 28
    8.2. Other Composition Operators 29
9. Conclusion 31

UNIVERSITY LIBRARIES
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA 15213

List of Figures

Figure 4-1: Arbiter and Two User Processes 8
Figure 4-2: Automaton for Trace set T_A 8
Figure 4-3: nMOS Mutual Exclusion Element (Mutex) 9
Figure 4-4: A Fair Arbiter 10
Figure 4-5: An Unfair Arbiter 11
Figure 5-1: Rendezvous Element and its Trace Set 13
Figure 5-2: Extended Automata for Arbiter Trace Sets 14

1. Introduction

1.1. Background

Arbiters and synchronizers continue to be an active topic of discussion and debate among computer scientists and designers. Much of the current interest in this area centers on new approaches to metastability and the synchronizer failure problem [2]; these approaches often involve or require asynchronous design methods. One such approach involves designing sampling and arbitration circuits that detect and indicate when they have exited metastability [9] [13]. When combined with an overall asynchronous design methodology, the resulting systems function correctly in the presence of arbitrary metastable behavior. Asynchronous circuits are also potentially faster than their synchronous counterparts, particularly in cases where computation complexity is data dependent. Finally asynchronous design methodologies cleanly separate correctness and speed concerns, thus removing the clock synchronization, distribution and skew difficulties that are becoming increasingly prevalent as the size of synchronous circuits increases.

Until recently the applicability of asynchronous design has been severely limited by the rapid increase of complexity with circuit size and the intricacies of race and hazard avoidance associated with classic Huffman-style design. In particular arbiters are difficult to design correctly [6]; for example two independent bugs have been found in a well-known asynchronous arbiter originally proposed by Seitz [1] [3] [14]. Trace theory is a communication-oriented time-independent formalism that is well suited to describing and specifying asynchronous speed-independent circuits; one of the main goals in the development of the theory and associated asynchronous design techniques has been to reduce the complexity of such design [11]. The concept of delay insensitivity is extremely useful in this regard; a delay-insensitive module's functionality is not affected by arbitrary delays in the interconnections between it and other system components [18]. A simple method of testing a trace structure specification of a module for delay insensitivity has recently been found; this and other work suggests that delay-insensitivity is an appropriate minimum requirement for interfaces among asynchronous self-timed modules, and has spurred interest in delay-insensitive systems [12].

Fairness, i.e. the impartial treatment of competing requests, is an important property desired of most arbiters; an unfair arbiter that favors one requestor to the exclusion of all others is trivial to implement and of little practical use. Fairness has already been extensively investigated by researchers working on the foundations of distributed systems; one of their important conclusions is that no single notion of fairness suffices for all applications. As a result the literature contains several precisely defined notions of fairness to cover the spectrum of possible uses [4] [5]. Inattention to the notions of

fairness being used, their precise definitions, and similar details have produced misleading results and caused needless confusion in this area.

1.2. Contributions of this Paper

A recent result claims to prove the non-existence of delay-insensitive fair arbiters [19]; this has potentially profound consequences for both the theory and practice of asynchronous design. The resulting controversy ranges from hardware designers who are sure such arbiters exist and therefore doubt the utility of the theory to theoreticians who wonder just exactly what the hardware designers have been building [15]. Among other goals, this paper makes a forthright attempt to settle the controversy.

We begin with an introduction to enough trace theory to formally present the concept of delay insensitivity and the above result. We then continue with a formal discussion of the possible notions of fairness and use them to analyze the non-existence result. This analysis reaches three important conclusions; first that the result is theoretically correct but of no practical significance because it uses an inappropriate notion of fairness, second that delay-insensitive fair arbiters do exist (which we show by exhibiting an example), and third that the present trace theory lacks the expressive power to specify any of the relevant notions of fairness for asynchronous self-timed arbiters.

Motivated by our third conclusion, the second section of this paper extends and generalizes trace theory to permit the expression of these fairness properties. Following an approach originally pioneered by Muller, we extend trace theory to infinite traces; as he discovered, liveness is necessarily important to the extension [10]. We discuss the impact and role of liveness in our extension by example. In the process of formulating the extension we develop a more general trace-theoretic composition operator that does not require the domain constraints (composability restrictions) used by other authors. Finally we define an extended concept of delay insensitivity and formally show that our fair arbiter from the first section is delay insensitive.

2. Trace Theory

Reasoning about the operation of an arbiter requires a formalism to describe the arbiter; we use trace theory for this purpose. The following presentation is an introduction; full details can be found elsewhere. [11]

2.1. Introduction

We consider the combination of a system and its environment which communicate with each other by sending signals. Signals are classified as input or output according to their direction at the system. For a hardware realization we would associate a signal with each wire with transmission accomplished by a level change. Our notion of signal differs in a fundamental manner from notions found elsewhere in the literature; sending of a signal is a unilateral action of the sender -- there is no notion of the receiver agreeing to the transmission of the signal (as in a CSP rendezvous) or of the receiver permitting or forbiddding the signal (as is the case for CCS actions) [8]. Such notions can be implemented using multiple signals for each such primitive; Martin [7] implements a CSP rendezvous using two signals, and the CCS action-response primitive requires at least three signals. As a result it is posible for unexpected signals to arrive at receivers (i.e. the receiver's specification does not include the possible arrival of the signal); trace theory regards such events as failures, and refers to them as instances of computation interference. An additional class of failures is due to our desire that signals closely model wires used to interconnect systems; level transitions which are too close together may cancel or interfere with each other causing the loss of signals. Such phenomena belong to the class of failures known as transmission interference. One of the major concerns of trace theory is the detection and sufficient conditions for the prevention of these classes of failures [16]. We will return to this shortly.

A trace structure is the specification of a system-environment pair. The set of symbols associated with all wires which pass between the system and its environment is the alphabet of the trace structure; it is partitioned into an input and an output alphabet. The remaining component of a trace structure is a trace set which specifies all possible sequences (or traces) of communications between the system and its environment. This is a subset of the set of all finite sequences of symbols from the trace structure's alphabet. A trace is a passive specification of a sequence of communicated symbols which has occurred; there is no notion of the sequence necessarily ending at the end of the trace. Therefore trace sets include traces that will necessarily be extended, traces that may be extended, and traces that will not be extended; these classes are not separated within a trace set. We use lower case letters from the beginning of the alphabet to denote symbols (a,b,c, etc.) and lower case letters from the end of the alphabet to denote traces (s,t,u, etc.). Formally we define:

> **Definition 1:** A trace structure *T* is a triple $\langle I,O,T \rangle$ where I and O are the disjoint input and output alphabets of *T*. Each is a finite set of symbols, and their union is denoted by A. T is the trace set, $T \subseteq (A)^*$. T must be prefix closed (i.e. for all traces s and t, $st \in T \rightarrow s \in T$).

Where more than one trace structure is involved, the structures and their components will be differentiated by primes and/or subscripts. The prefix-closure requirement above is motivated by the

intention that a trace set specify all possible communication sequences for the system-environment pair. A corollary is that the empty trace ε must belong to every trace structure; this is ensured by our prefix closure requirement.

2.2. Delay Insensitivity

Delays in communications among systems can adversely affect the operation of some systems; clock distribution networks provide excellent examples of the measures often taken to limit these delays. One of the tenets of asynchronous design is that systems should function correctly in the presence of arbitrary delays; this property, called delay-insensitivity, has the potential of abstracting the correctness of hardware design away from timing considerations. To formalize this concept we must consider arbitrary delays between the system and its environment. One consequence of these delays is that the system and its environment may not always be able to agree on the order in which signals occur. As a result during operation there are two traces generated, one at the system and one at the environment; due to delays these need not be the same. For correctness we need to ensure that if one of these traces can be extended by transmitting a symbol, the trace at the other end can be extended by receiving the symbol. Our formal definition of delay insensitivity for trace structures (see below) combines this property with the absence of transmission interference. The correct operation of a system specified by such a trace structure is not affected by arbitrary delays in wires connecting the system to its environment.

It is useful to think of a delay-insensitive system as being encased in a foam rubber wrapper with flexible and changing boundaries. In this model delay-insensitive systems have identical specifications at both sides of the wrapper. Therefore the presence of a trace in a trace set may necessitate the presence of other traces to preserve delay-insensitivity. It is also necessary to exclude traces which represent transmission interference. These properties can be captured in the following rules due to Udding [18].

> **Definition 2: Delay Insensitivity:** A trace structure $\mathcal{T} = \langle I,O,T \rangle$ is delay insensitive iff it satisfies the following rules. For these rules the type of a symbol a is either input (if $a \in I$) or output (if $a \in O$).
>
> **Rule 0:** The wrapper can reverse the order of two concurrent signals travelling in the same direction, so a DI trace set must be closed under this transformation. The rule is: for all traces s,t and all symbols a,b of the same type $sabt \in T$ iff $sbat \in T$.
>
> **Rule 1:** The wrapper can arbitrarily delay the transmission of a signal; hence one side of the system-mechanism pair cannot change its readiness to receive a signal from the other by transmission of a second signal -- the signal to be disabled may already be on its way through the wrapper. Since this must be true at both ends, the formal rule is: for all traces s and symbols a and b of different types $sa \in T \wedge sb \in T \rightarrow sab \in T$.

Rule 2: The wrapper can reverse the order of two concurrent signals travelling in opposite directions. The definition of concurrency is not easy in this case because two concurrent signals of different types can generate traces that look like causal relationships at one or both both boundaries. [i.e. input before output at system and output before input at the environment boundary] The obvious analogue to rule 0 is too restrictive; if symbol a causes symbol b, then the trace sabt is in the trace set whereas the trace sbat is not. If the signals are actually concurrent, then a response which depends on the second signal at one boundary must be compatible with the trace that has the two signals in the other order at the other boundary. Formally the rule is: for all traces s,t and symbols a,b,c where b is of a different type than a and c $sabtc \in T \wedge sbat \in T \rightarrow sbatc \in T$.

Rule 3: Finally we assume that the delays through the wrapper are inertial, i.e. two transitions on one wire which are too close together could cancel. To prevent this we require: for all traces s and symbols a $saa \notin T$.

Udding has shown that delay-insensitivity is a sufficient condition for the absence of transmission and computation interference under a basic notion of composition. [18]

3. A Simple Example

For the purposes of elucidation we now turn to a simple problem; is it possible to build a fair two-input delay-insensitive arbiter under the condition that requests cannot be withdrawn after they are made? A typical application of such an arbiter is a situation in which two independent processes are competing for exclusive access to a shared resource. We realize the arbiter interface by request inputs a, b and corresponding grants p, q. Without loss of generality we assume that these symbols are also used to signal and acknowledge the release of the resource being arbitrated for. Thus the trace apap represents a complete cycle of resource use; the first a requests the resource, the first p grants it, the second a releases the resource, and the second p acknowledges release of the resource and signals the arbiter's readiness to accept new requests. The interfaces are independent, i.e. the first process (a,p interface) does not see the symbols of the second interface (b,q).

We have already noted that in a delay-insensitive interface, one side of the interface cannot unilaterally disable a symbol generated by the other side (rule 1 above); thus for a delay insensitive arbiter once a request is enabled (at the start or after a release acknowledgement) it may occur at any future point. Similarly a grant may be followed by a release at any future point.

An additional requirement on our arbiter is that it not unnecessarily starve a process, i.e. in the absence of requests from the other process, requests from the first process will always be accepted and granted. With the symbols used for our example this means that both $(ap)^*$ and $(bq)^*$ must be subsets of the trace set of the arbiter-world interface.

3.1. Udding's Non-Existence Result

Udding considers the arbiter we have presented in the example and takes up the issue of whether there is such an arbiter that is both fair and delay-insensitive. Having already defined the notion of delay insensitivity, we now consider fairness. Udding reasons that our example arbiter is unfair if it always grants requests to one process in the face of an outstanding request from the other and formalizes this in trace theory by defining such an arbiter to be unfair if its trace set contains the traces $b(ap)^k$ for all positive k. [19]

Using this definition Udding was able to prove that any delay-insensitive trace structure for our example represents an unfair arbiter, and therefore concluded that fair delay-insensitive arbiters do not exist. [19] This result is not altogether surprising; since the process interfaces are independent, any b request must be concurrent with both a and p -- therefore the wrapper is free to change the order of b with respect to a and p, thus allowing it to be arbitrarily delayed.

To put this result in perspective, we now examine the topic of fairness and the various notions thereof.

4. Fairness

4.1. Fairness and Arbiters

There are many definitions of fairness in the literature [4] [5]; we briefly outline some of the more important ones in the context of an arbiter:

1. Weak Fairness (Justice, Response to Insistence). If a request is continually asserted, it will eventually be granted. Equivalently any request is either infinitely often granted or infinitely often unasserted (disabled). [Example: at a fast food restaurant if you stand in line continuously, you will eventually reach the counter and be served.]

2. Strong Fairness (Fairness, Response to Persistence) If a request is asserted infinitely often, it is granted infinitely often. Equivalently any request is either infinitely often granted or continually unasserted after some point. (Often expressed as disabled almost everywhere, i.e. in an infinite sequence of choice sets presented to the arbiter an almost everwhere disabled request appears in at most finitely many.) [Example: at a fast food restaurant, even if you have to leave the line occasionally to go to the bathroom, you will still eventually be served -- perhaps someone saves your place in line.]

3. Response to Impulse. A single request to the arbiter will eventually be granted. [Example: Airplane boarding; obtaining a boarding pass (single request) assures you of getting on the plane, but there is no guarrantee that the plane will be boarded in the order that boarding passes are issued.]

4. Strict Fairness. The order in which requests are granted corresponds strictly to the order

in which they are made. [Example: this corresponds to 'take a number' systems; people are served in the order of their numbers (requests).]

For conciseness we use i.o. to abbreviate infinitely often, and a.e. to abbreviate almost everywhere.

4.2. Fairness and the Example

Returning to the issue of fairness in connection with our example, we begin by considering how Udding's definition of unfairness is related to the standard definitions of fairness. If we negate his definition of unfairness, the result is the weakest definition of fairness consistent with his original definition of unfairness; this definition specifies that an arbiter is fair if there is a positive integer k for which $b(ap)^k$ is not it its trace set. (and similarly for not starving 'a' requests) Since trace structures specifying behavior must be prefix-closed, we apply prefix closure to this definition and obtain: $\exists k > 0$ such that $\forall m \geq k$ $b(ap)^m$ is not in the trace set of the arbiter.

In other words for an arbiter to be fair (i.e. fail to be unfair according to the definition) there must be a fixed bound (2k - 2) on the number of symbols which occur in any trace between a request and its grant or the end of the trace. This notion of fairness is intermediate in strength between Strict Fairness and Response to Impulse; we call it Bounded Fairness. Similar concepts often arise in the implementations of concurrent systems.

Our concern here is with the concept of delay-insensitivity; Bounded Fairness is not a delay-insensitive notion. This is because the wrapper could delay the b request long enough to allow too many a requests to be granted. Udding's proof shows this formally by application of the definition of delay-insensitivity. We also note that Strict Fairness is not delay-insensitive for the same reason.

For the example under consideration the other three notions of fairness are identical because requests cannot be withdrawn.

Theorem 3: If requests cannot be withdrawn, then the concepts of Weak Fairness, Strong Fairness, and Response to Impulse are equivalent.

Proof: Due to the relative strengths of the general concepts, it suffices to show that Weak Fairness implies Response to Impulse.

Weak Fairness → Response to Impulse: Under Weak Fairness a request asserted continuously is granted. Since requests cannot be withdrawn, a single assertion of a request causes it to be asserted continuously until granted. Therefore any single request is always eventually granted.(Response to Impulse) □

4.3. A Fair Arbiter

We now consider whether we can build a fair arbiter for this notion of fairness. To begin with we need a trace structure which specifies the arbiter; a finite automaton accepting the trace set of such a structure, T_A is shown below. All states are final in this automaton because T_A is prefix-closed.

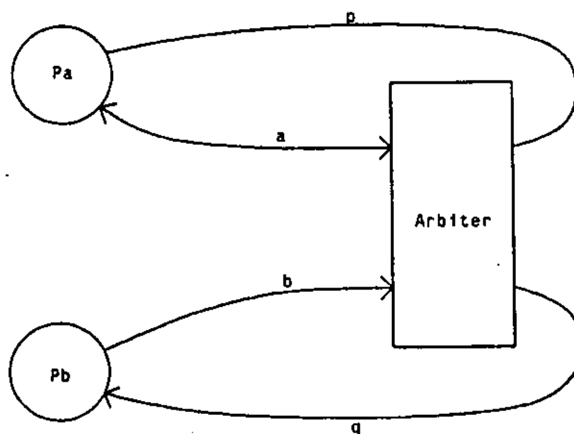

Figure 4-1: Arbiter and Two User Processes

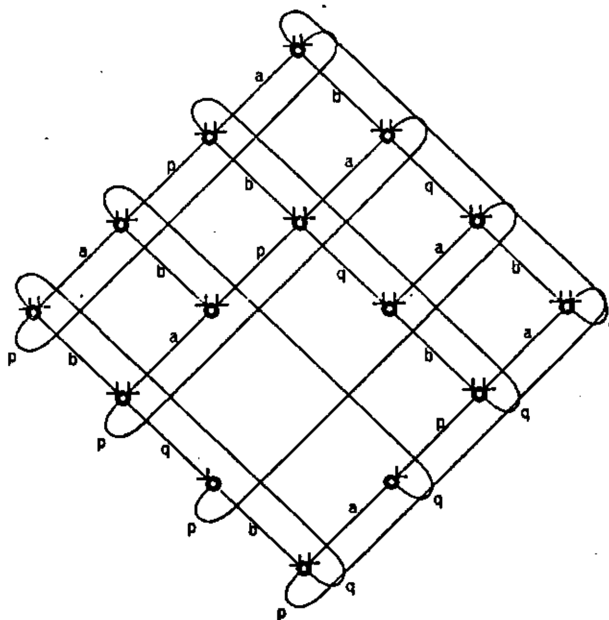

Figure 4-2: Automaton for Trace set T_A

We note that this trace structure has disjoint input (ab) and output (pq) alphabets, and is prefix closed. The remaining rules for delay insensitive trace structures can be verified by inspection of the automaton:

0: The only concurrent symbols in this structure are a and b, as p and q can never occur adjacent in any trace. Inspection of the automaton reveals that the symbols a and b commute at all places where they are allowed by the automaton, therefore this rule holds.

1: The only possible violation of this rule occurs at the state reached by trace ab, but p and q are of the same type (output) so the rule is not violated.

2: This trace structure satisfies a stronger condition which implies the holding of Rule 2, namely that when the order of two adjacent symbols in a trace can be reversed, these symbols actually commute. [i.e. if the symbols are x and y, then for all traces s such that both sxy and syx are in the trace set sxy and syx correspond to the same state in the minimal automaton accepting the trace set.] Hence if sbat $\in T_A$ then for all c sabtc $\in T_A$ iff sbatc $\in T_A$; therefore the rule holds.

3: The automaton rejects all traces with repeated characters, hence no trace of the form saa is an element of T_A.

Therefore this trace structure is delay-insensitive. Our fair arbiter is based on the nMOS mutual exclusion element described by Seitz [13] which corresponds to the trace structure T_A. This element and the arbiter are shown below.

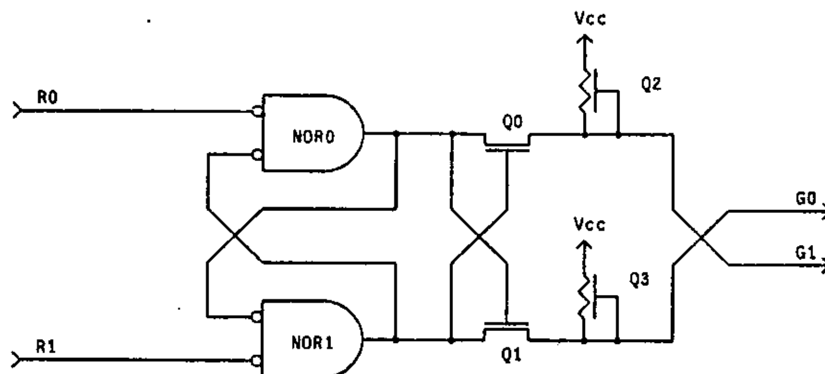

Figure 4-3: nMOS Mutual Exclusion Element (Mutex)

The delays are crucial to the fairness of the arbiter. To understand how these delays are designed

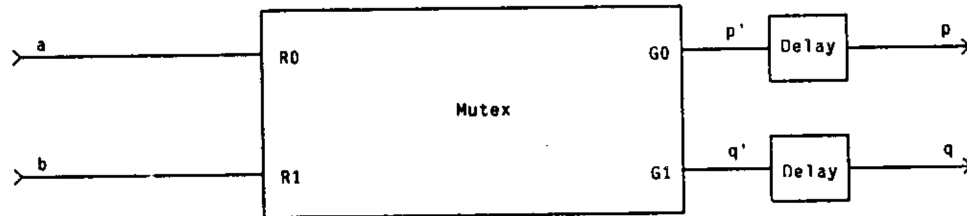

Figure 4-4: A Fair Arbiter

consider the behavior of the mutual exclusion element when R0 is released in the presence of an outstanding request on R1. The initial conditions are:

- R0 - 0. R1 - 0.
- NOR0 - 1. NOR1 - 0.
- Q0 - off. Q1 - on.
- G0 - 0 [due to Q1]. G1 - 1 [pulled up by Q2]

Now R0 becomes 1. This causes NOR0 to become 0 which in turn sets off two sequences of events:

- Q1 turns off allowing Q3 to pull G0 high (to a 1 condition) acknowledging the release of the resource.
- NOR1 becomes 1 turning on Q0 which causes G1 to come low issuing a grant to process 1.

These sequences are concurrent, and there is no guarantee that the second one will finish; if NOR1 is slow, process 0 could see the ack and make a new request that flips NOR0 back to a 1 before NOR1. To ensure fairness we demand that the second sequence finishes before an acknowledge is issued. To do this we insert the delays to prevent either process from seeing an ack too early. Since there is no metastability problem in this situation, the worst case propagation time for the second event sequence can be calculated [NOR gate, pass transistor, NOR gate pulls down pulled up output]; a delay of this magnitude ensures that the second process is always issued a grant in this case.

Showing fairness of this arbiter is now straightforward. We assume that each process eventually releases the resource after any grant to it. Consider a request from the second process (b). After some finite (but unspecified) delay this request arrives at the arbiter. There are now 4 cases to consider:

1. The resource is free (p and q are high, and no a request arrives during the propagation time for the NOR gates) -- The resource is immediately granted to the second process.

2. The arbiter is granting the resource (including the possibility of a metastable state caused in part by this request) -- Eventually the arbiter finishes granting [including falling out of the metastable state]. Either the second process receives the grant, or the first process receives the grant. In the latter situation we are reduced to the next case.

3. The resource is in use -- Eventually the other process releases the resource; we showed above that this causes a grant to the second process.

4. The resource is being released -- this reduces either to the first case or the second case (if the other process makes a new request very quickly).

In all cases we eventually reach either 1 or 3 above which causes a grant to be issued to the second process. By symmetry we conclude that any single request is always granted, and hence this arbiter is fair. Since this arbiter is also delay-insensitive, we therefore conclude that fair delay-insensitive arbiters do exist.

4.4. An Unfair Arbiter

For comparison we now present an unfair arbiter. This is also based on the nMOS mutual exclusion element, and uses two of them; one performs the mutual exclusion, and the other is used as a random bit generator by having its inputs wired together in such a way that it enters a metastable state when a falling edge is applied. The circuit for this arbiter is shown below.

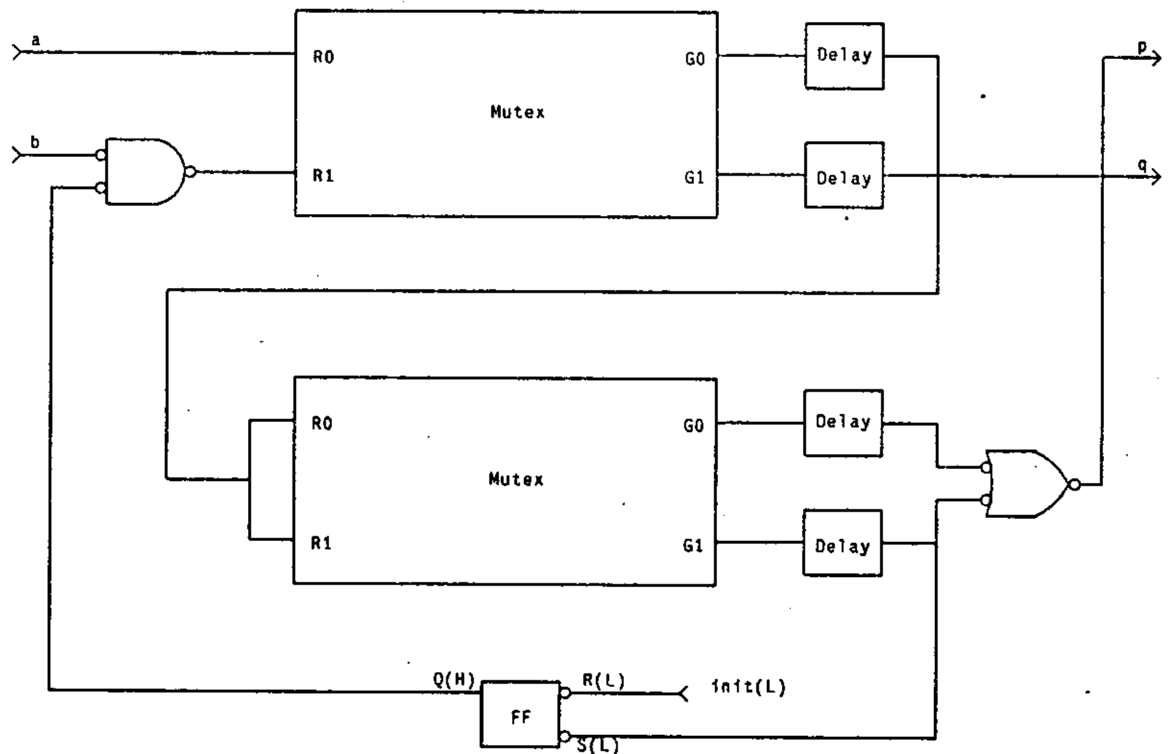

Figure 4-5: An Unfair Arbiter

This arbiter is clearly unfair; once the random bit generator generates a 1 the next b request is never

granted. The surprising property of this arbiter is embodied in the following theorem:

Theorem 4: The unfair arbiter has exactly the same trace structure as the fair arbiter.

Proof: The alphabets are identical, so it remains to show equality of the trace sets; we do this by showing mutual containment.

(fair $\subseteq$ unfair): The traces of the fair arbiter correspond to traces of the unfair arbiter in which the random bit generator has not generated a 1.

(unfair $\subseteq$ fair): The traces of the unfair arbiter for which the random bit generator has not generated a 1 correspond to the traces of the fair arbiter. If a 1 has been generated there are two cases; either a b request is outstanding at the end of the trace or it isn't. In the former case this corresponds to a fair arbiter trace in which the b request is delayed by the 'foam rubber wrapper', the latter case corresponds directly to a fair arbiter trace as the disable signal is not visible to the external interface. □

This shows that the existing trace theory of finite traces is not adequate for treating the concept of fairness; both the fair and unfair arbiter have exactly the same trace structure, namely T_A. Hence it is meaningless to talk about an arbiter specified solely by such a trace structure as fair or unfair. We now proceed to discuss an extension to trace theory which is powerful enough to deal with the concept of fairness.

5. Infinite Traces and Trace Theory

Because a trace theory whose trace sets consist only of finite traces is not powerful enough to represent fairness and related concepts, we now consider trace sets which may contain infinite traces. Most of the trace sets encountered in the theory of finite traces are regular, and often conveniently represented by the finite automata which accept them. For our extended theory, most of the trace sets will be regular in the general sense, a related concept which we now define:

Definition 5: Regular in the General Sense: A trace structure T is regular in the general sense iff there exists an extended automaton which accepts exactly T.

Definition 6: Extended Automaton: An extended automaton is a six-tuple $(Q,\Sigma,\delta,q_0,F,\pi)$ where:

- Q is the set of states.
- Σ is the input alphabet.
- $\delta: Q\times\Sigma \rightarrow Q$ is the transition function.
- $q_0 \in Q$ is the initial state.
- $F \subseteq Q$ is the set of final states.
- $\pi \subseteq \mathcal{P}(Q)-\emptyset$ is the set of sets of preferred states.

Definition 7: Acceptance of Finite Strings: We extend δ to finite strings in the usual way. $[\delta^*(q,as) = \delta^*(\delta(q,a),s)]$. A finite string s is accepted by the extended automaton iff $\delta^*(q_0,s) \in F$.

Definition 8: Acceptance of Infinite Strings: For an infinite string $w = a_1a_2\ldots$ define $\sigma(w)$ to be the sequence $\langle q_0,\delta(q_0,a_1),\delta^*(q_0,a_1a_2),\ldots\rangle$; this is the sequence of

states the automaton encounters in reading w. Define $I(w) = \{q \mid q$ occurs infinitely many times in $\sigma(w)\}$. The extended automaton accepts w iff $I(w) \in \pi$.

Our motivation for using this extended notion of regularity is the following theorem, originally proved by Muller [10]:

Theorem 9: For a digital network N composed of finitely many components, let S(N) be the total set of finite and infinite traces that can be produced by the network. Then S(N) is regular in the general sense.

5.1. Example

The canonical simple example used to illustrate concepts in asynchronous systems is the Rendezvous element. In this tradition we illustrate the use of an extended automaton as a specification for a rendezvous element with inputs a,b and output c. The operation of such an element is to wait for inputs on both a and b, and then send an output on c. A general automaton that specifies it is $R = (Q_R, \Sigma_R, \delta_R, q_{0R}, F_R, \pi_R)$ where

- $Q_R = \{1,2,3,4\}$
- $\Sigma_R = \{a,b,c\}$
- δ_R is shown below
- $q_{0R} = 1$
- $F_R = Q_R$
- $\pi_R = \{ \{1,2,4\}, \{1,3,4\}, \{1,2,3,4\} \}$

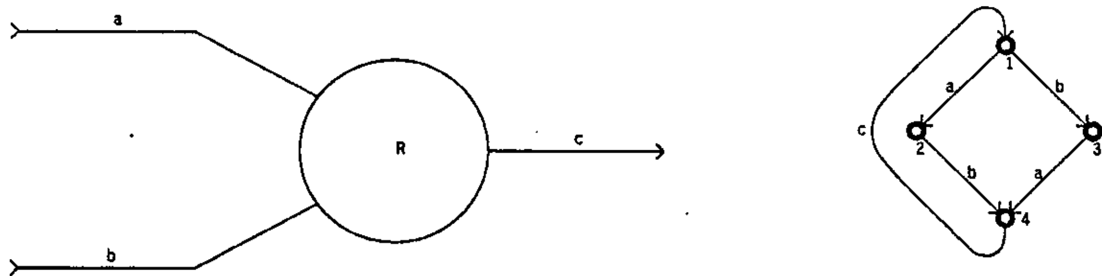

Figure 5-1: Rendezvous Element and its Trace Set

π_R contains three sets of preferred states. The corresponding sets of infinite traces are:

- $\{1,2,4\}$: $((ab \mid ba)\, c)^* (abc)^\omega$
- $\{1,3,4\}$: $((ab \mid ba)\, c)^* (bac)^\omega$
- $\{1,2,3,4\}$: $((abc)^+ (bac)^+)^\omega \cup ((bac)^+ (abc)^+)^\omega$

where a^+ is shorthand for $a;a^*$. An interesting observation here is that these three disjoint sets of infinite traces have identical sets of finite prefixes.

5.2. Arbiter Automata

One indication of the power of this formalism is that it is able to distinguish the fair and unfair arbiters, which the set of finite traces formalism was unable to do. As a result the extended automata for the fair and unfair arbiter differ only in their π components; we present the common components before taking up their differences. In the following F designates the automaton for the fair arbiter and U designates the automaton for the unfair arbiter.

- $Q_F = Q_U = \{1,2,\ldots,10,12,13,\ldots,16\}$
- $\Sigma_F = \Sigma_U = \{a,b,p,q\}$
- $\delta_F = \delta_U$; see diagram below
- $q_{0F} = q_{0U} = 1$
- $F_F = F_U = Q_F = Q_U$.

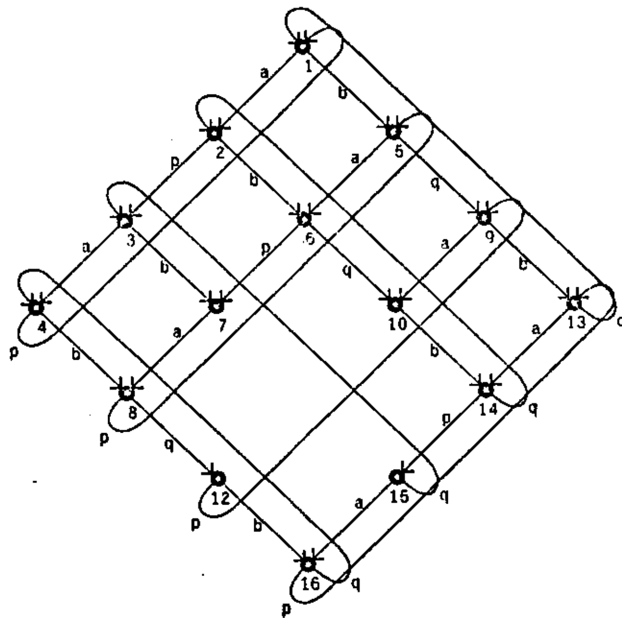

Figure 5-2: Extended Automata for Arbiter Trace Sets

To make defining the π's easier, we divide the sets of preferred states into the following 7 classes corresponding to different behaviors:

1. {1,2,3,4} -- B never makes a request after some finite point.
2. {5,6,7,8} -- B is starved; makes a request which is never granted.
3. {13,14,15,16} -- B is blocked; some b release is never acknowledged.
4. {1,5,9,13} -- A never makes a request after some finite point.
5. {2,6,10,14} -- A is starved; makes a request which is never granted.
6. {4,8,12,16} -- A is blocked; some a release is never acknowledged.
7. This class contains all sets of states which form a strongly connected subgraph

containing at least one state from each diagonal row and column. ex. {1,5,9,13,14,15,16}, {1,2,5,6,7,8,12,13,16} -- Both processes have requests granted infinitely often.

For the fair arbiter π_F is the union of classes 1, 4, and 7. For the unfair arbiter π_U is the union of classes 1, 2, 4, and 7. Thus the unfair arbiter may exhibit unfair behavior (class 2), whereas the fair arbiter is forbidden from doing so. For future reference we define the following trace structures:

Definition 10: Arbiter Trace Structures:

Fair Arbiter -- $A_F = \langle\{a,b\},\{p,q\},T_{AF}\rangle$ where T_{AF} is the set accepted by automaton F above.

Unfair Arbiter -- $A_U = \langle\{a,b\},\{p,q\},T_{AU}\rangle$ where T_{AU} is the set accepted by automaton U above.

Process a -- $P_a = \langle\{p\},\{a\},\underline{pref}((ap)^*)\rangle$

Process b -- $P_b = \langle\{q\},\{b\},\underline{pref}((bq)^*)\rangle$

6. Composition Operator

One of the goals of this work is to formalize the definition of delay insensitivity for trace structures which contain infinite traces. One possibility is to extend the rules given earlier. We prefer to define delay insensitivity in terms of a composition operator which reflects how the modules specified or described by trace structures behave when composed. Two potentially useful operators have been defined by other authors in the pursuit of different goals. [16] [12] We have been able to unify and generalize these approaches in a single composition operator which also applies to trace structures which contain infinite traces. A final benefit of this work is that the desired definition of delay-insensitivity has been achieved, and is identical to the existing one (rules) on structures whose trace sets contain only finite traces.

6.1. Notational Conventions

For clarity we employ the following notational conventions in this paper:

- If a trace is written as a sequence of trace symbols, e.g. stu, then only the last symbol (u) may denote an infinite trace. A trace ending in a simple symbol, e.g. ta, is always finite.
- We freely extend functions defined on singletons to sets. If f is defined on singletons a and Σ is a set of such singletons, then $f(\Sigma) = \bigcup_{a \in \Sigma} f(a)$.
- Further we consider symbols to be traces of length 1, thus allowing the application of functions defined on trace sets to sets of symbols.
- Finally in many cases both a trace or set of traces and the containing structure are

arguments to an operator; we use a colon notation to indicate this and a subscript convention to abbreviate it. (i.e. the symbol t_1 is shorthand for $t_1:T_1$, T_1 is shorthand for $T_1:\mathcal{T}_1$ etc.)

6.2. Directed Traces

In ordinary (undirected) traces a symbol represents the instantaneous occurance of its transmission and receipt. In order to directly include potential wire delays in our behavioral modes we need to separate these two events. Accordingly each undirected symbol is replaced by two directed symbols, one for each event. The directed traces upon which our composition operator will be based consist solely of directed symbols.

Definition 11: Directed Symbols: A symbol is directed if it is postfixed with either ! or ?. Otherwise it is undirected.

The postfixed ! corresponds to transmission, and the postfixed ? corresponds to reception. We now define the notions of direction and undirection for traces and trace structures.

Definition 12: Direction and Undirection of Traces: Consider a trace structure $\mathcal{T}$. For $t \in T$, the directed trace corresponding to t, D(t), is obtained by replacing every $a \in I$ in t with a? and every $b \in O$ with b! . D has no effect on directed symbols occurring in t.

Similarly the undirected trace corresponding to t, U(t), is obtained by replacing every a? in t with a and every b! with b. U has no effect on undirected symbols occurring in t.

Definition 13: Direction and Undirection of Trace Structures: for a trace structure $i(T) = \langle I,O,T \rangle$ define $\mathfrak{D}(\mathcal{T}) = \langle D(I),D(O),D(T) \rangle$. Also note that $D(A) = D(I) \cup D(O)$.

Similarly define $\mathfrak{U}(\mathcal{T}) = \langle U(I),U(O),U(T) \rangle$ and note that $U(A) = U(I) \cup U(O)$.

These definitions are sufficiently general to apply to trace structures containing partially directed traces.

6.3. Wire Trace Structures

One of the major contributions of this paper is the introduction of a new class of trace structures. The existing trace theory concentrates on the behavior of modules and systems, and thus each trace structure represents such a component. Based in part on the foam rubber wrapper model of system interconnection used earlier, we believe that properties of the transmission media are as important as properties of the individual systems. The most that has been done previously in this area is the separation of symbol transmission and reception as in the previous section. We go further and introduce a new class of trace structures, the wire trace structures, to explicitly represent properties of the interconnection media. In this section we present examples of wire modules corresponding to such trace structures that exhibit different transmission properties. This contrasts with previous approaches in which these media properties are implicit in the composition operator definition.

Our new wire trace structure serve as the interconnecting 'glue' for ordinary structures; they are distinguished by the presence of 'output' symbols such as a! in their directed input alphabets and likewise the presence of 'input' symbols such as a? in their output alphabets. [Note: a wire trace structure has either both a? and a! in its alphabet or neither.] These structures exist only as directed structures; applying our $\mathcal{U}$ operator to such a structure produces a structure whose input and output alphabets are identical. The resulting structure is not valid because we require that the input and output alphabets be disjoint. Wire trace structures can often be parametrized by the symbols transmitted along the wires that the structure represents; for the following definitions Σ denotes a set of undirected symbols.

Definition 14: Ordinary Wire Module For a set of symbols Σ, we define $W[\Sigma] = \langle I_W, O_W, T_W \rangle$ where

- $I_W = \{a! \mid a \in \Sigma\}$
- $O_W = \{a? \mid a \in \Sigma\}$
- $T_W = \{t \mid t\lceil\{a?,a!\} \in \underline{pref}[(a!a?)^\omega]$ for all $a \in \Sigma$ and every symbol in t is either a? or a! for some $a \in \Sigma\}$

where $\underline{pref}$ is the prefix closure operator and $\lceil$ denotes projection. Note that the final condition in the definition of T_W [Only a? or a! for $a \in \Sigma$ may occur in t] is implicit in the definitions of I_W and O_W; we may omit such conditions in susequent definitions.

This trace structure captures the intuitive notion of a wire as an ideal transmission medium; a symbol presented at one end (a!) is transmitted to the other (a?) before the next is presented. [Note that this does not specify that the wires have no delay, but rather it excludes delay-line behavior of wires.] Although this is a useful behavior subset, real wires exhibit such phenomena such as delay line behavior and noise. The latter does not concern us; we regard it as an analog concern that should be addressed and solved at that level. On the other hand, delay and delay-line phenomena are of material concern to us because one of our goals is the specification of systems that are insensitive to delays. Therefore the underlying model must include delay-dependent behavior; we include such behavior in our model via the G (General Wire) modules. The synchronization trace sets [16] play an important role in its definition. For our purposes we do not need them in their full generality; the following definition suffices:

Definition 15: Synchronization Trace Set: For symbols b,c and $k \geq 0$ we define $\Psi_k(b,c) = \{ x \in (b|c)^* \mid \text{for all finite prefixes y of x, } 0 \leq \#_b(y) - \#_c(y) \leq k\}$

where $\#_a(t)$ is the number of times the symbol a occurs in the trace t. The k in the above definition captures the potential lead of b over c; for example the trace set of a semaphore is $\Psi_1(P,V)$. A wire with unknown delay-line behavior can allow any lead of a! over a? -- this is formalized as follows:

Definition 16: General Wire Module: For a set of symbols Σ we define $G[\Sigma] = \langle I_G, O_G, T_G \rangle$ where:

- $I_G = \{a! \mid a \in \Sigma\}$
- $O_G = \{a? \mid a \in \Sigma\}$
- $T_G = \{t \mid$ for all finite prefixes x of t and all $a \in \Sigma$ there exists $k \geq 0$ such that $x \lceil \{a?, a!\} \in \Psi_k(a!, a?)\}$

Defining the trace set in terms of prefixes is necessary to allow unbounded leads of a! over a? within a single infinite trace. The reader may note the obvious correspondence between G[{a}] and an infinite stack with a! as the push operation and a? as the pop operation. It follows that G modules are not regular in the general sense. One of the advantages of delay-insensitivity is that W modules (which are regular in the general sense) may be substituted for G modules without changing system behavior.

6.4. Operator Definition

An advantage of defining wire trace structures is that we can separate the definition of the composition operator from the properties of the 'wire' interconnections; wire trace structures are treated identically to other trace structure arguments of the composition operator.

Our composition operator consists of two main stages, a weaving stage which directs the symbols and merges the trace sets, and a resolution stage which produces all the undirected behaviors corresponding to the set of directed behaviors formed in the previous stage.

The directed weave operators combine the trace sets presented as arguments into a single trace set. The combination is similar to interleaving, but matches directed symbols that occur in more than one trace structure's directed alphabet. This matching corresponds to the direct connection between the end of a wire and the module transmitting or receiving the corresponding symbol. For disjoint alphabets, directed weaving reduces to interleaving.

Definition 17: Directed Weave Operators:

For trace structures $T_1, \ldots, T_n$, define $W_d(t_1, \ldots, t_n) = \{s \mid \forall i\ s \lceil D(A_i) = D(t_i)\}$

$\mathcal{W}_d(T_1, \ldots, T_n) = \langle I_d, O_d, T_d \rangle$ where

- $I_d = \bigcup_i D(I_i)$
- $O_d = \bigcup_i D(O_i)$
- $T_d = W_d(T_1, \ldots, T_n)$

Since all our wires have precisely two ends, we make a connectability restriction on the directed alphabets of the trace structures presented to the directed weave and composition operators: for every symbol a, both a? and a! may appear at most once as input symbols, and at most once as output symbols. Notice that the result of the directed weave operator may violate this condition; this is deliberate, and will be resolved by the remaining stages of the composition operator.

The resolution stage of our composition operator takes a directed trace and forms all the undirected traces that correspond to it. In a directed trace the transmission of a symbol corresponds to an a!a? pair possibly separated by other symbols. Since we imagine each module to be encased in a foam rubber wrapper, the corresponding a symbol in the undirected trace must be placed at the boundary between the wrappers. This boundary could be at the originating module, the receiving module, or anywhere in between; hence the a could be placed at the a! position, the a? position, or anywhere in between. To allow for the dynamic variations in our foam rubber wrapper model we do not fix this location, but let it vary from instance to instance. Unmatched a!'s are assumed to be part of a!a? pairs in which the a? is yet to occur, thus the corresponding a can be placed at the a! or anywhere subsequently in the trace. We capture this formally in the following definitions due mainly to Schols [12]:

Definition 18: Immediate Resolution: For a trace t (which may contain both directed and undirected symbols) s is an immediate resultant of t [written s $\underline{ir}$ t] iff there exist a,x,y,y',z such that $(y,y')\lceil\{a?\} = \emptyset$ and

$$((s = xyay'z) \wedge (t = xa!yy'a?z)) \vee$$
$$((s = xyay') \wedge (t = xa!yy'))$$

Definition 19: Resolution of Traces and Trace Structures: Define $\underline{r}$ to be the reflexive transitive closure of $\underline{ir}$. Then we define the resultant of a trace t belonging to a trace structure $T = \langle I,O,T \rangle$ to be $R(t) = \{s \mid s\ \underline{r}\ t \text{ and } s\lceil D(A) = \emptyset\}$.

For a trace structure $T = \langle I,O,T \rangle$ $\mathfrak{R}(T) = \langle U(I),U(O),R(T) \rangle$

Resolution of traces is a stepwise operation in which partially directed traces occur as intermediaries; it may be thought of as occurring in a trace structure which contains both directed and undirected symbols in its alphabet. The $\mathfrak{I}$ and $\mathfrak{U}$ operators are sufficiently general to handle such structures.

For a closed system (in which all symbols which appear appear as inputs, outputs and wires), it suffices to resolve the directed weave, as $\underline{ir}$ enforces exactly the desired relationship between a trace symbol and its corresponding inputs and outputs. Intuitively we may think of this relationship as follows:

1. A symbol corresponding to an input-output pair may occur at any point in the subtrace delimited by that pair.
2. A symbol corresponding only to an output may occur at any point in the subtrace beginning at the output.
3. A symbol corresponding only to an input is nonsense. [i.e. inputs are not spontaneously generated.]

For systems which are not closed, i.e. cases in which we wish to compose trace structures to derive an external specification, the last two rules cause serious problems. This is due to the absence of

external inputs and outputs when deriving an external specification; R eliminates such inputs and allows migration of outputs arbitrarily far forward, yielding bizarre results. Correcting this requires explicit attention in the composition operator definition; we achieve this by defining an externalization operator which undirects such symbols, thus preventing R from manipulating them.

Definition 20: Externalization Operators: For a trace structure $\mathcal{T} = \langle I,O,T \rangle$ where I and O need not be disjoint define the external symbol operator S_e as:

$$S_e(\mathcal{T}) = \{a? \mid a! \notin A\} \cup \{a! \mid a? \notin A\}$$

E is the externalization operator on traces, $E(t,S) = t'$ where t' is derived from t by replacing all symbols in S with their undirected counterparts.

For convenience define U_s, the selective undirect operator as: $U_s(A,B) = A-(A \cap B) \cup U(A \cap B)$

Finally for the externalization operator on trace structures let $S = S_e(\mathcal{T})$: $\mathcal{E}(\mathcal{T}) = \langle U_s(I,S), U_s(O,S), E(T,S) \rangle$

Our composition operator produces a trace set from a trace structure; as with Milner's approach in CCS [8] we believe that defining composition in this manner is the most general setting, and the resulting separation of concerns between composition and symbol hiding/restriction is both useful and productive. In a later section we will describe how other operators (ex. projection) may be composed with our operator to yield other composition operators useb by other authors.

Definition 21: Composition Operator:

$$C(t_1, \ldots, t_n) = \underline{pref}(R(\mathcal{E}(\mathcal{W}_d(t_1, \ldots, t_n))))$$

The prefix closure operator is necessary for full generality on non-delay-insensitive trace structures.

For closed systems this operator produces the behavior obtained by using the specified wires to interconnect the specified trace structures. For systems that are not closed (i.e. cases in which we are composing component specifications to obtain an external specification), there are six possibilities based on the presence of a! and a? in the input and output alphabets:

1. a! as output : The E operator converts all such outputs to a in place. This corresponds to outputs occurring at the boundary of the module producing them.

2. a? as input : The E operator converts all such inputs to a in place. This corresponds to the allowed positions of inputs at the module accepting such inputs.

3. a? as input and a! as output : This is an internal symbol for which no wire module was specified. The R operator implicitly supplies a G[a] module (the most general case) for this symbol.

4. a? as output and a! as input : This is a dangling wire; i.e. a wire module was specified that doesn't connect to anything. The R operator allows occurrances of a as specified by the wire module.

5. a? as both input and output, a! as input : This is an external input signal with an attached wire; inputs are allowed wherever the wire can transmit them to be legally received by the connected module (as specified by its trace structure). This may be too general, as was alluded to in the section on the externalization operators.

6. a? as output,a! as both input and output : This is an external output with an attached wire; the results are similar to the previous case.

6.5. Composability

We have already stated the only restriction we place on traces and trace structures to which our composition operator is applied; namely that in the collection of directed alphabets each directed symbol can appear at most once as input and at most once as output. This is a major advantage of our new operator over existing operators; previous work has often required complicated definitions of trace composability that had to be satisfied by any traces presented to the composition operator. Neither of the definitions we are familiar with from previous work immediately extend to infinite traces, and both are somewhat unintuitive. A major advantage of our composition operator is that it does not require an independent definition of composability.

Here are the two previous definitions of composability:

Definition 22: Udding-Composability: For trace structures T,T' and traces t,t' from those structures,

$$\begin{aligned} comp(t,t') = &\ (t = \varepsilon \wedge t' = \varepsilon) \vee \\ &(\exists\, a,t_0 :\ t = t_0 a \wedge comp(t_0,t') \wedge (a \in O' \rightarrow \#_a(t') > \#_a(t_0)) \vee \\ &(\exists\, b,t'_0 :\ t' = t'_0 b \wedge comp(t,t'_0) \wedge (b \in O \rightarrow \#_b(t) > \#_b(t'_0)) \end{aligned}$$

where $\#_a(t)$ is the number of times the symbol a occurs in the trace t.

Definition 23: Schols-Composability: For trace structures T,T' and traces t,t' from those structures,

$$\begin{aligned} c(t,t') = &\ (\forall\, a \in (O \cap I')\ \#_a(t) \geq \#_a(t')) \wedge (\forall\, b \in (I \cap O')\ \#_b(t') \geq \#_b(t)) \\ &\wedge t\ \underline{nodeadlock}\ t' \end{aligned}$$

where

$$\begin{aligned} &t\ \underline{nodeadlock}\ t' = \\ &(\forall\, a,b,t_0,t'_0 \text{ such that } a \in (O \cap I'),\ b \in (I \cap O'),\ t_0 b \in \underline{pref}(t),\ t'_0 a \in \underline{pref}(t'), \\ &\#_a(t_0) > \#_a(t'_0) \wedge \#_b(t'_0) > \#_b(t_0)) \end{aligned}$$

We now prove that our operator does not require an independent definition of composability.

Theorem 24: For trace structures T,T' and finite traces t,t' from those structures, the following are equivalent:

- c(t,t')

- comp(t,t')
- $C(t,t',G[A \cap A']) \neq \emptyset$
- $C(t,t') \neq \emptyset$

Proof: $c(t,t') \equiv comp(t,t')$ was established by Schols. [12] The equivalence of the last two conditions is a consequence of the following lemma:

Lemma: Let DS(t) be the set of directed symbols occurring in t, and let T_g be the trace set component of G[U(DS(t))]. Then $R(t) \neq \emptyset$ iff $t\lceil DS(t) \in T_g$.

Proof: If $R(t) = \emptyset$ then there is at least one symbol a for which ir cannot match a?'s with a!'s. Hence $R(t\lceil\{a!,a?\}) = \emptyset$. In the reverse direction if for all symbols a we have $R(t\lceil\{a!,a?\}) \neq \emptyset$, then $R(t) \neq \emptyset$ by the same reasoning. Therefore it suffices to establish this lemma for one symbol, i.e. for DS(t) = {a!,a?}. Since the definition of G[] restricts only the finite prefixes of traces, it suffices to establish this property for finite t. [i.e. this property also holds for infinite traces.]

($\rightarrow$): From the definition of R, $R(t) \neq \emptyset$ iff every a? in t is preceded by a matching a!. Hence it follows that for all finite prefixes x of t $x\lceil DS(t) \in \Psi_k(a!,a?)$ where $k = \#_{a!}(x)$, and therefore $t\lceil DS(t) \in T_g$.

($\leftarrow$): From the definitions of G[] and Ψ_k, $t\lceil DS(t) \in T_g$ iff every a? in t is preceded by a matching a!. Hence it follows that $R(t) \neq \emptyset$. □

This lemma formally shows that the composition operator supplies G modules for unconnected wires; observing that $A \cap A'$ is exactly the set of symbols corresponding to unconnected wires in (T,T') yields the claimed equivalence (of the last two conditions above) immediately.

We now proceed to establish the middle equivalence for finite traces by induction on the length of the traces. We show the proof for an increase in the length of t; the proof for an increase in the length of t' can be obtained by syntactic substitution.

Base: $|t| = |t'| = 0$ iff $t = t' = \varepsilon$. comp(t,t') is true by definition. $C(\varepsilon{:}T,\varepsilon{:}T',\varepsilon{:}G[A \cap A']) = \varepsilon$ hence $C(\varepsilon{:}T,\varepsilon{:}T',\varepsilon{:}G[A \cap A']) \neq \emptyset$. The desired conclusion follows immediately.

Inductive Step: ($\rightarrow$): Since we have an increase in the length of t, comp(t,t') is equivalent to:

$$\exists\, a,t_0 \text{ such that } t = t_0 a \wedge comp(t_0,t') \wedge (a \in O' \rightarrow \#_a(t') > \#_a(t_0))$$

The inductive hypothesis applies to t_0 and t', hence $comp(t_0,t')$ iff $C(t_0,t',G[A \cap A']) \neq \emptyset$ iff $W_d(t_0,t',G[A \cap A']) \neq \emptyset$ because $\emptyset$ is preserved by $R(\mathcal{S}())$.

Now assume comp(t,t') and let $s \in W_d(t_0,t',G[A \cap A'])$ such that $R(\mathcal{S}(s)) \neq \emptyset$ and let $s_r \in R(\mathcal{S}(s))$. [i.e. $s_r \in C(t_0,t',G[A \cap A'])$]. Also let $w = s\lceil A_G$ where A_G is the alphabet of $G[A \cap A']$. We represent this trace structure as the tuple $\langle I_G,O_G,T_G\rangle$. There are three cases to consider:

1. $a \in A - A'$. Let $\alpha = D(a)$. Since $a \notin A'$ and $a \notin A_G$ it follows that

$s\alpha \in W_d(t,t',w{:}G[A \cap A'])$

Applying $R(\mathcal{S}())$ to both sides we conclude

$s_r a \in R(\mathcal{S}(\mathcal{W}_d(t,t',w{:}G[A \cap A'])))$

and therefore

$C(t,t',G[A \cap A']) \neq \emptyset$

as was to be shown.

2. $a \in (O \cap I')$: The definition of T_G allows any finite member trace to be extended by a!; therefore wa! $\in T_G$, and so

   $sa! \in W_d(t,t',wa!{:}G[A \cap A'])$

   Applying $R(\mathcal{S}())$ to both sides we conclude

   $s_u a \in R(\mathcal{S}(\mathcal{W}_d(t,t',wa!{:}G[A \cap A'])))$

   and therefore

   $C(t,t',G[A \cap A']) \neq \emptyset$

   as was to be shown.

3. $a \in (I \cap O')$: Since $\#_a(t') > \#_a(t_0)$, therefore $\#_{a!}(w) > \#_{a?}(w)$, hence wa? $\in T_G$ by definition. It now follows that

   $sa? \in W_d(t,t',wa?{:}G[A \cap A'])$

   Because s has the same excess of a! over a? as w, $R(s_e a?) \neq \emptyset$ where s_e is the result of externalizing s, furthermore $s_r \in R(s_e a?)$. [Obtaining s_r from s_e requires ir'ing one or more unmatched a!. Replacing one of these by a match with the final a? obtains the same result from $s_e a?$.] So applying $R(\mathcal{S}())$ to both sides we conclude

   $s_r \in R(\mathcal{S}(\mathcal{W}_d(t,t',wa?{:}G[A \cap A'])))$

   and therefore

   $C(t,t',G[A \cap A']) \neq \emptyset$

   as was to be shown.

($\leftarrow$): In the reverse direction we assume $C(t,t',G[A \cap A']) \neq \emptyset$, hence $W_d(t,t',G[A \cap A]) \neq \emptyset$. Therefore let $s \in W_d(t,t',G[A \cap A'])$ such that $R(\mathcal{S}(s)) \neq \emptyset$ and the last symbol of s is $\alpha = D(a)$ where a is the last symbol of t. [Such an s always exists, else we are in the case of an increase in length of t'] Now we have $s = s_0\alpha$ and $t = t_0 a$. Also let $w = s\lceil A_G$.

Lemma: $R(\mathcal{S}(s_0)) \neq \emptyset$. There are two possiblities, either $\alpha = a!$, or $\alpha = a?$. In the first case the final α is undirected by an application of the second clause of ir; s_0 can be undirected by omitting this application in the undirection of s. In the second case α is undirected by an application of the first clause of ir; s_0 can be undirected by substituting an application of the second clause of ir for this application in the undirection of s. Therefore every undirection of s yields a corresponding undirection of s_0. □

Based on the lemma let $s_{0r} \in R(\mathcal{S}(s_0))$. Now there are two cases to consider:

1. $a \in (A - A')$: Since $\alpha \notin D(A')$ and $\alpha \notin D(A_G)$ it follows that

   $s_0 \in W_d(t_0,t',w{:}G[A \cap A'])$

Applying $R(\mathcal{E}())$ to both sides we conclude

$$s_{0r} \in R(\mathcal{E}(\mathcal{W}_d(t_0,t',w:G[A \cap A'])))$$

Therefore $C(t_0,t',G[A \cap A']) \neq \emptyset$. This is equivalent to $comp(t_0,t')$ by the inductive hypothesis. Since $a \notin O'$ we conclude:

$$\exists\, a,t_0 \text{ such that } t = t_0 a \wedge comp(t_0,t') \wedge (a \in O' \rightarrow \#_a(t') > \#_a(t_0))$$

as was to be shown.

2. $a \in (A \cap A')$: From the definitions of w, W_d and G[] it follows that α is the last symbol of w, hence we have $w = w_0\alpha$. Because $\alpha \notin D(A')$ prefix closure allows us to conclude:

$$s_0 \in W_d(t_0,t',w_0:G[A \cap A'])$$

Applying $R(\mathcal{E}())$ to both wides we conclude

$$s_{0r} \in R(\mathcal{E}(\mathcal{W}_d(t_0,t',w_0:G[A \cap A'])))$$

Therefore $C(t_0,t',G[A \cap A']) \neq \emptyset$. This is equivalent to $comp(t_0,t')$ by the inductive hypothesis. If $a \in O'$, then since $w_0 a? \in T_G$, it follows that $\#_{a!}(w_0) > \#_{a?}(w_0)$ and hence $\#_a(t') > \#_a(t_0)$, so we conclude

$$\exists\, a,t_0 \text{ such that } t = t_0 a \wedge comp(t_0,t') \wedge (a \in O' \rightarrow \#_a(t') > \#_a(t_0))$$

as was to be shown. □

7. Liveness

7.1. Example

With the basic structure of a trace theory which includes infinite traces in hand, we can now return to our original problem; specification of a fair delay-insensitive arbiter. One indication of success in this area would be the following property involving the arbiter trace structures we defined previously:

$$C(P_a,P_b,A_F,W[\{a,b,p,q\}]) = T_{A_F} \qquad (1)$$

This states that a system including a fair arbiter exhibits only fair behaviors; unfortunately this is not the case because the composition in property 1 produces the trace $abp(ap)^{\omega}$ which is not in T_{A_F} as follows:

$(a!p?)^{\omega} \in \mathfrak{I}(P_a)$ $\qquad\qquad$ $b! \in \mathfrak{I}(P_b)$

$w = a!a?b!p!p?(a!a?p!p?)^{\omega} \in W[\{a,b,p,q\}]$

$w\lceil\{a!,a?\} = (a!a?)^{\omega}$ $\qquad\qquad$ $w\lceil\{b!,b?\} = b!$

$w\lceil\{p!,p?\} = (p!p?)^{\omega}$ $\qquad\qquad$ $w\lceil\{q!,q?\} = \epsilon$

$(a?p!)^{\omega} \in \mathfrak{I}(A_F)$

$a!a?b!p!p?(a!a?p!p?)^{\omega} \in \mathcal{W}_d((a!p?)^{\omega}, b!, w, (a?p!)^{\omega})$

$\mathcal{E}$ has no effect as the system is closed

$abp(ap)^{\omega} \in R(a!a?b!p!p?(a!a?p!p?)^{\omega})$, therefore
$abp(ap)^{\omega} \in C(P_a, P_b, A_F, W[\{a,b,p,q\}])$

Examination of the above reveals that this was caused by wire failure; the signal b was transmitted along the wire (b!), but was never received. Therefore the arbiter behaved correctly (because it never saw the request), but nonetheless starved process B. Hence in order to formally specify a fair arbiter, we must specify wires that do not fail in this manner; such an absence of failure is a liveness property. This simple example was sufficient to demonstrate failure of a wire module; more complicated examples can be constructed in which ordinary (non-wire) modules fail in similar ways producing equally unexpected results. Without the power to specify liveness of modules, it is impossible to specify fairness properties.

7.2. Changes to the Theory

To remedy this we propose changing trace theory to explicitly include an appropriate notion of liveness. (i.e. our trace specifications will implicitly exclude the failure mode mentioned above) We justify this by claiming that live systems are of far more interest than those exhibiting the pathological behavior of unexpected death at any moment. We also do not wish to incorporate any notions of fault tolerance beyond those implicit in delay insensitivity; such extended notions are not appropriate to a basic model of hardware behavior. On an intuitive level the liveness property we desire can be roughly stated as: "If a trace can be extended by an output symbol (from any module), then the trace will eventually be extended (but not necessarily by an output symbol)." This ensures that once a module is enabled to do something, it either eventually does something, or is instructed to do something else. This condition is weak enough to apply to non-delay-insensitive systems such as missing pulse detectors. [A missing pulse dectector watches a constant incoming pulse train and produces a pulse every time a pulse is missing from the train. An infinite pulse train without missing pulses constantly cycles the detector through states where it is enabled to produce an output, but no output is ever produced] For delay-insensitive systems our condition is equivalent to the stronger condition that any trace which can be extended by an output is eventually extended by an output.

An important underlying concept here is that of a module being enabled; we formalize this in terms of complete and incomplete traces. Incomplete traces correspond to conditions under which the module is enabled; complete traces correspond to all other conditions, i.e. complete behaviors of the module including infinite behaviors.

Definition 25: Complete Traces: For a trace structure $T = \langle I,O,T \rangle$, all infinite traces are complete. A finite trace $t{:}T$ is complete iff $\forall\, a \in O,\ ta \notin T$. Otherwise t is incomplete.

As an example, consider the rendezvous element presented earlier; all infinite traces, and all finite traces which do not end with ab or ba are complete. The incomplete traces are those ending with ab or ba because such traces and only such traces can be extended by an output, c. Incomplete traces represent potential liveness failures of the type described above; we can exclude them by preventing their composition into infinite traces. The following projection operator accomplishes this:

Definition 26: New Projection Operator: For a trace s and trace structure $T = \langle I,O,T \rangle$, $s \downarrow T$ =

- $s \lceil A$ if s is finite and $s \lceil A \in T$.
- $s \lceil A$ if s is infinite and $s \lceil A \in T$ and $s \lceil A$ is complete.
- $\varnothing$ otherwise.

Notice that this has no effect if the involved trace structures contain only finite traces, as then only the first and third cases above apply. To complete this section we present the reformulated definitions of the wire modules and the directed resultant operator obtained by substituting this projection operator for the previous one. We note that the definition of this new operator in terms of projection onto a trace structure requires an additional case in the wire module definitions.

Definition 27: Directed Weave Operator -- Revised Definition:

For trace structures $T_1, \ldots, T_n$ define $W_d(t_1, \ldots, t_n) = \{s \mid \forall\, i\ (s \downarrow \mathfrak{I}(T_i) = D(t_i))\}$

Definition 28: Ordinary Wire Module -- Revised Definition For a set of symbols Σ, we define $W[\Sigma] = \langle I_W, O_W, T_W \rangle$ where

- $I_W = \{a! \mid a \in \Sigma\}$
- $O_W = \{a? \mid a \in \Sigma\}$
- $T_W = \{t \mid t \in \underline{pref}[(a!a?)^{\omega}]\}$ for $\Sigma = \{a\}$
- $T_W = W_d(W[\{a_1\}], \ldots, W[\{a_n\}])$ for $\Sigma = \{a_1, \ldots, a_n\}$

Definition 29: General Wire Module -- Revised Definition: For a set of symbols Σ we define $G[\Sigma] = \langle I_G, O_G, T_G \rangle$ where:

- $I_G = \{a! \mid a \in \Sigma\}$
- $O_G = \{a? \mid a \in \Sigma\}$
- $T_G = \{t \mid$ for all prefixes x of t there exists $k \geq 0$ such that $x \in \Psi_k(a!,a?)$ $\}$ for $\Sigma = \{a\}$
- $T_G = W_d(G[\{a_1\}], \ldots, G[\{a_n\}])$ for $\Sigma = \{a_1, \ldots, a_n\}$

Finally we need to make a slight change to the resolution operator; the present definition of $\underline{ir}$ allows unmatched outputs to be undirected in all traces. For infinite traces such outputs represent liveness

failures; if we prevent ir from undirecting them, then R() will fail also; this causes implicit G modules supplied by R() to satisfy our liveness condition. The following revised definition of ir accomplishes this:

Definition 30: Immediate Resolution -- Revised Definition: For a trace t (which may contain both directed and undirected symbols) s is an immediate resultant of t [written s ir t] iff there exist a,x,y,y',z such that $(y,y')\lceil\{a?\} = \emptyset$, y and y' are finite traces, and

$((s = xyay'z) \wedge (t = xa!yy'a?z)) \vee$
$((s = xyay') \wedge (t = xa!yy'))$

This completes our modifications; notice that this has not changed any of the operators or definitions for structures consisting only of finite traces.

We now demonstrate the effectiveness of the liveness modification by proving property 1 under these revised definitions. Our proof does not depend explicitly on the wire module specified; it holds for both the W and G modules.

Theorem 31:

$C(P_a,P_b,A_F,W[\{a,b,p,q\}]) = T_{A_F}$
$C(P_a,P_b,A_F,G[\{a,b,p,q\}]) = T_{A_F}$

Proof: The right to left containment is obvious; for any trace in T_{A_F} appropriate traces for the structures on the left side can be constructed by inspection. For finite traces the left to right containment is a consequence of delay-insensitivity of the finite arbiter trace structure. This leaves the case of left to right inclusion for infinite traces. We will prove this by showing that C preserves the class of the A_F trace on the left side. There are three cases:

1. Class 1: These traces contain an even number of b's with a matching number of q's. The liveness condition in $\mathcal{W}_d$ requires exactly the same number of b's and q's to appear in the trace resulting from C; such an infinite trace is in class 1.

2. Class 4: Identical argument to previous case on symbols a and p instead of b and q.

3. Class 7: These traces contain infinitely many of all 4 symbols; therefore the resulting trace must contain infinitely many of all 4 due to the liveness condition in $\mathcal{W}_d$. Such an infinite trace is in Class 7.

The above argument implicitly uses the fact that the 7 classes cover all the possible infinite traces of the arbiter. Delay-insensitivity of the finite structure is a sufficient condition to conclude that any left-side trace must belong to one of these classes. □

8. Applications of the Composition Operator

8.1. Delay Insensitivity

Schols [12] defines delay insensitivity in terms of a composition operator for closed systems, ©, that is essentially pref ∘ $\mathcal{U}$ ∘ $\mathcal{W}_d$ with changes to produce reasonable alphabets. [$\mathcal{U} \circ \mathcal{W}_d$ puts all symbols of a closed system in both input and output alphabets.] Formally for structures T and T' such that $I = O'$ and $O = I'$ we have:

$T © T' = \langle I, O, \underline{pref}(R(\mathcal{W}_d(T, T'))) \rangle$

Since such a system is closed, $\mathcal{S}$ has no effect; therefore

$$T © T' = \langle I, O, C(T, T') \rangle \tag{2}$$

We now turn to defining delay insensitivity in terms of composition operators. The following theorem forms the basis for our definition:

Theorem 32: For trace structures whose trace sets contain only finite traces the following are equivalent:

- Udding's rules for delay insensitivity.
- $T = T © \bar{T}$ and $\forall$ a, saa $\notin T$
- $T = C(T, \bar{T})$ and $\mathcal{W}_d(T, \bar{T}, G[A]) = \mathcal{W}_d(T, \bar{T}, W[A])$

where $\bar{T}$ is obtained from T by interchanging the input and output alphabets.

Proof: The equivalence of the first two conditions was established by Schols and Verhoeff. [12] [20] An immediate consequence of equation 2 above is that

$T = T © \bar{T}$ is equivalent to $T = C(T, \bar{T})$

We are therefore left to establish the equivalence of the second conjuncts in the last two conditions given that the first conjunts hold; we do this by establishing the equivalence of their negations. Assume without loss of generality that a is an output symbol of T

saa $\in$ T $\rightarrow$

$D(s{:}T)a!a! \in \mathfrak{D}(T) \wedge D(s{:}\bar{T})a?a? \in \mathfrak{D}(\bar{T}) \rightarrow$

$s_d a!a!a?a? \in C(T, \bar{T}, G[A]) \rightarrow$

$\mathcal{W}_d(T, \bar{T}, G[A]) \neq \mathcal{W}_d(T, \bar{T}, W[A])$

where s_d is obtained from s by replacing every symbol b in s with the pair b!b? and the last implication follows from the fact that $s_d a!a!a?a? \downarrow W[A] = \varnothing$.

In the reverse direction if substitution of a G module for a W module changes $\mathcal{W}_d$, then some trace t in $\mathcal{W}_d(T, \bar{T}, G[A])$ allows a! to lead a? by more than one. Since T, $\bar{T}$, and G[A] are prefix closed, $\mathcal{W}_d$ of them is also; therefore consider the trace t_0 which is the prefix of t up to and including the first a! symbol that produces a lead of two. [i.e. $\#_{a!}(t_0) = \#_{a?}(t_0) + 2$] One of the possible undirects of t_0 is a trace of the form saa, since both unmatched a! may be undirected at the end of the trace. Hence for some a saa $\in C(T, \bar{T})$, so the first conjunct gives us saa $\in$ T as claimed. □

We note that the C operator is not powerful enough to express the absence of transmission interference (no traces of the form saa) in a satisfactory manner; an operator such as $\mathcal{W}_d$ which separates the transmission and reception of a symbol is needed. On the basis of the above theorem we adopt the third equivalent condition above as the definition of delay insensitivity for trace structures which may contain infinite traces:

Definition 33: Delay Insensitivity -- Infinite Trace Structures:

A trace structure $T = \langle I,O,T \rangle$ which may contain infinite traces is delay-insensitive iff:

$$T = C(T,\bar{T}) \text{ and } \mathcal{W}_d(T,\bar{T},G[A]) = \mathcal{W}_d(T,\bar{T},W[A])$$

For trace structures whose finite counterparts are delay-insensitive, it is sufficient to check only the first condition (i.e. $T = C(T,\bar{T})$) as the second condition fails for finite prefixes iff it fails for the entire structure. (i.e. absence of transmission interference is a finite property.)

Having come this far, we now show that the trace structure for our fair arbiter is delay-insensitive.

Theorem 34: A_F is delay insensitive.

Proof: Since A_F as a finite trace structure is delay-insensitive it suffices to show that $C(A_F,\bar{A}_F) = T_{A_F}$.

The left to right inclusion is obvious as the required traces can be constructed by inspection in all cases. The right to left inclusion follows from Theorem 31 and the fact that $\bar{A}_F \subseteq C(P_a,P_b)$. □

8.2. Other Composition Operators

We are aware of four other composition operators for trace structures, the blend, weave [11] [17], agglutinate [16]; and composite [12]. Having already taken up the composite, we show in this section that our C operator also generalizes the other 3 operators by defining agglutinate in terms of C and exhibiting a wire module that causes C to become a weave operator. (The blend operator is a functional composition of the weave operator and a projection operator).

The simplest of these operators are the blend and weave operators; weaving produces a trace set from trace structures.

Definition 35: Weave: For trace structures $T = \langle I,O,T \rangle$ and $T' = \langle I',O',T' \rangle$, the weave of T and T' is

$$T\underline{w}T' = \{ x \mid x\lceil A \in T \wedge x\lceil A' \in T' \}$$

The blend operator is the weave composed with a projection away from the common symbols.

Definition 36: Blend: For trace structures T and T', the blend of T and T' is

$$T\underline{b}T' = \langle I - O' \cup I' - O, O - I' \cup O' - I, T\underline{w}T'\lceil(A \% A') \rangle$$

where % is the symmetric difference operator.

We now consider how the C operator is related to the blend. It is productive to view the blend operator as a model in which signal transmission and reception happen synchronously and instantaneously. We can therefore recreate it by defining a wire module that exhibits this behavior.

Definition 37: Dummy Wire Modules: $D_d[\Sigma] = \langle I_d, O_d, T_d \rangle$ and $D_w[\Sigma] = \langle I_w, O_w, T_w \rangle$ where

- $I_d = I_w = \{a! \mid a \in \Sigma\}$
- $O_d = O_d = \{a? \mid a \in \Sigma\}$
- $T_d = \{t \mid$ There exists a trace s of which t is a prefix such that $s = x_1 x_2 \ldots$ and $\forall i$ $\exists a \in \Sigma\ x_i = a!a?\ \}$
- $T_w = \{t \mid t = x_1 x_2 \ldots x_n$ or $t = x_1 x_2 \ldots$ where $\forall i\ \exists a \in \Sigma\ x_i = a!a?\ \}$

Traces in a dummy wire module are sequences of the form a!a?b!b? etc., i.e. each transmitted symbol is received before any other symbol is transmitted. We note that not only is this a stricter restriction than that for the ordinary wire module, and that it does not distribute over the set of symbols, unlike the G and W wire modules. [i.e. $W[A \cup B] = \mathcal{W}_d(W[A], W[B])$, but this property does not hold for D modules.] The key property of D modules is the following:

$T \underline{w} T' = C(T, T', D_w[A \cap A'])$

The proof is obvious and left to the reader. Substitution of a D_d module for the D_w module in the above yields the directed weave operator defined by Ebergen [17].

We now examine the agglutinate operator; for trace structures T and T', Snepscheut defines their agglutinate as:

$T \underline{g} T' = T?!(I \cap O', O \cap I')\ \underline{b}\ G[A \cap A']\ \underline{b}\ T'?!(I' \cap O, O' \cap I)$

where $T?!(X,Y)$ directs symbols in $X \subseteq I$ as inputs and $Y \subseteq O$ as outputs, and $\underline{b}$ is the blend operator. This differs from our approach in that only the common symbols are directed, whereas we direct all symbols and then selectively undirect the external symbols; the results are the same in both cases because symbols appearing in only one trace alphabet (external symbols) cannot be matched with any symbols in another trace structure by either the blend or directed resultant operators. We note the following properties:

$T?!(I \cap O', O \cap I') = \mathcal{E}(\mathcal{I}(\langle I \cup I', O \cup O', T \rangle))$

$T'?!(I' \cap O, O' \cap I) = \mathcal{E}(\mathcal{I}(\langle I \cup I', O \cup O', T' \rangle))$

Substituting these into the above definition, and noting that $\mathcal{E}$ and $\underline{b}$ commute we obtain

$T \underline{g} T' = \mathcal{E}(\mathcal{I}(T)\ \underline{b}\ G[A \cap A']\ \underline{b}\ \mathcal{I}(T'))$

where the alphabet changes have been dropped from the arguments to the $\mathcal{I}$'s because $\underline{b}$ supplies the correct alphabets to $\mathcal{E}$. Since $\lceil D(A \% A') \circ \mathcal{W}_d = \underline{b} \circ \langle \mathcal{I}, \ldots, \mathcal{I} \rangle$ and $\mathcal{I}$ is an identity operator on wire modules, we can conclude that:

$T \underline{g} T' = \mathcal{E}(\mathcal{W}_d(T, G[A \cap A'], T') \lceil D(A \% A'))$

Reversing the order of the projection and $\mathcal{S}$ requires changing the projection target to its undirected version as D(A % A') is precisely the set of symbols undirected by $\mathcal{S}$ above, therefore

$$T \underline{g} T' = \mathcal{S}(\mathcal{W}_d(T, G[A \cap A'], T'))\lceil(A \% A')$$

The directed symbols remaining in traces produced by $\mathcal{S}$ above are all in the alphabet of the G module. Hence R cannot produce the empty set for any such trace, and furthermore inserting R between $\mathcal{S}$ and the projection does not change the result because R does not affect the relative order of external symbols (A % A') and all others vanish in the projection. Therefore

$$T_{T\underline{g}T'} = R(\mathcal{S}(\mathcal{W}_d(T, G[A \cap A'], T')))\lceil(A \% A')$$

For prefix closed trace structures $\underline{g}$ produces prefix-closed trace structures, consequently from the definition of C we obtain the following:

$$T \underline{g} T' = \langle(I - O') \cup (I' - O), (O - I') \cup (O' - I), C(T, T', G[A \cap A'])\rangle$$

where the projection away from the symbols common to T and U is implicit in the alphabets of the right hand side.

9. Conclusion

In this paper we have considered the existence of and formally specifying delay-insensitive fair arbiters. We have shown that the exact notion of fairness used is of critical importance because some of the common notions are not delay-insensitive. Further we have shown that for the relevant notions of fairness, the existing trace theory of finite traces lacks sufficient expressive power to adequately specify a fair delay-insensitive arbiter. [i.e. the specification of a fair arbiter is also satisfied by an unfair arbiter.] Based on this we have extended trace theory to include infinite traces, and shown by example the importance of including liveness in such a theory. The extended theory is sufficiently expressive to distinguish fair arbiters from unfair ones, and we have use it to exhibit a delay-insensitive fair arbiter, thus establishing their existence. In addition our extended theory generalizes the existing trace theory by introducing a composition operator(C) that at once generalizes the existing operators and obviates the composability restrictions used by previous authors. Finally our extended theory introduces wire modules as an abstraction to capture the important role transmission media properties play in circuit behavior.

References

[1] Bochman, G. V.
Hardware Specification with Temporal Logic: An Example.
IEEE Transactions on Computers C-31(3), March, 1982.

[2] Chaney, T. J. and C. E. Molnar.
Anomalous Behavior of Synchronizer and Arbiter Circuits.
IEEE Transactions on Computers C-22(4):421-422, April, 1973.

[3] Dill, D. L., and E. M. Clarke.
Automatic Verification of Asynchronous Circuits using Temporal Logic.
In *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 127-144. 1985.

[4] Gabbay, D., A. Pnueli, S. Shelah, and J. Stavi.
On the Temporal Analysis of Fairness.
In *Symposium on Principles of Programming Languages*, pages 163-173. 1980.

[5] Lehman, D., A. Pnueli, and J. Stavi.
Impartiality, Justice and Fairness: The Ethics of Concurrent Termination.
Technical Report CS81-16, Weizmann Institute of Science, Department of Applied Mathematics, July, 1981.

[6] Martin, A. J.
A delay-insensitive Fair Arbiter.
Technical Memorandum 5193:TR:85, Computer Science Department, California Institute of Technology, Pasadena, CA, July. 1985.

[7] Martin, A. J.
The Design of a Self-Timed Circuit for Distributed Mutual Exclusion.
In *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 245-260. 1985.

[8] Milner, R.
Lecture Notes in Computer Science. Volume 92: *A Calculus of Communicating Systems.*
Springer-Verlag, 1980.

[9] Molnar, C. E., T.-P. Fang, and F. U. Rosenberger.
Synthesis of Delay-Insensitive Modules.
In *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 67-86. 1985.

[10] Muller, David E.
The General Synthesis Problem for Asynchronous Digital Networks.
In *Conference Record of the Eigth Annual Symposium on Switching and Automata Theory.* 1967.

[11] Rem, Martin et. al.
Trace Theory and the Definition of Hierarchical Components.
In *3rd Caltech Conference on VLSI*, pages 225-239. 1983.

[12] Schols, H.
A Formalisation of the Foam Rubber Wrapper Principle.
Master's thesis, Eindhoven University of Technology, 1985.

[13] Seitz, C. L.
System Timing.
Introduction to VLSI Systems.
Addison-Wesley, Reading, Massachusetts, 1980, Chapter 7.

[14] Seitz,C. L.
Ideas About Arbiters.
LAMBDA , First Quarter, 1980.

[15] Seitz, C. L.
Personal Communication.

[16] v.d. Snepscheut, J.L.A.
Trace Theory and VLSI Design.
PhD thesis, Eindhoven University of Technology, 1983.

[17] Sproull, B., I. Sutherland, and C. Molnar.
Seminar on Self-Timed Systems.
Course Notes.
Carnegie-Mellon University, Spring 1985.

[18] Udding, J.T.
Classification and Composition of Delay-Insensitive Circuits.
PhD thesis, Eindhoven University of Technology, 1984.

[19] Udding, J.T.
On the non-existence of delay-insensitive fair arbiters.
Technical Memorandum 306, Computer Systems Laboratory, Washington University, St. Louis, MO, July, 1985.

[20] Verhoeff, T. and H. Schols.
Delay-insensitive Directed Trace Structures Satisfy the Foam Rubber Wrapper Postulate.
Computing Science Notes 85/04, Department of Mathematics and Computing Science, Eindhoven University of Technology, August, 1985.