EINDHOVEN UNIVERSITY OF TECHNOLOGY

Partially ordered computations, with applications to VLSI-design

Citation for published version (APA):
Rem, M. (1983). Partially ordered computations, with applications to VLSI-design. In J. W. Bakker, de, & J. Leeuwen, van (Eds.), *Foundations of Computer Science IV, part 2* (pp. 1-44). (Mathematical Centre Tracts; Vol. 159). Stichting Mathematisch Centrum.

Document status and date:
Published: 01/01/1983

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 04. Oct. 2023

PARTIALLY ORDERED COMPUTATIONS, WITH APPLICATIONS TO VLSI DESIGN

M. Rem
Eindhoven Technological University, Eindhoven, the Netherlands

1. INTRODUCTION

VLSI (Very Large Scale Integration) is a medium for the execution of computations. Like all integrated circuits, a VLSI circuit consists of transistors, pads, and connections between them. The pads are points via which the communication with the circuit's environment takes place. A transistor can best be compared with an on-off switch. Over the years the transistors in integrated circuits have become smaller, thus allowing more transistors to be put on the same silicon chip. Making a transistor smaller shortens its switching delay. In VLSI chips the switching delays are so short compared to the delays in connections that the latter can no longer be ignored. The properties of the VLSI medium that are relevant to our exposition will be discussed in Section 3.

A general-purpose computer is also a medium for the execution of computations. In what respects is the VLSI medium different from traditional implementations? What makes VLSI an interesting subject? There are four differences we would like to point out:

(1) VLSI is a concurrent medium: it offers a surface of thousands, and in the future possibly millions, of simultaneously active computing elements.

(2) VLSI is a two-dimensional medium. The computing elements that together constitute a component have to be laid out in the plane. In a traditional implementation the computation has to be mapped onto a one-dimensional store. We have good techniques of achieving the latter, but the difference between one and two dimensions is the difference between lining up and solving a jigsaw puzzle.

(3) VLSI allows only limited connectivity. If a value computed in one place has to be used in a different place, a connection between the two places

must be present. But the VLSI medium does not allow an arbitrary placement of connections.

(4) The VLSI medium is not order-preserving. The connections in a circuit exhibit delays. When signals are sent via different connections from one place to another they can, of course, not be received before they are sent. But they may be received in an order that is different from the order in which they were sent.

There is a fifth problem, but that one is not unique to VLSI design. It is a problem that permeates all of computing science: complexity bridling. Uncontrolled concurrency causes uncontrolled complexity, even for moderately sized components. The well-known technique for avoiding complexity consists in partitioning - also known as modularizing - the components into subcomponents. Subcomponents are again components. The whole component thus exhibits a tree structure. Such tree-structured components are known as hierarchical components. Given the specifications of the subcomponents and the way in which the subcomponents constitute the component, we must be able to deduce the specification of the whole component. In view of the central role of complexity bridling, hierarchical composition will be used all through these notes.

A hierarchical design method not only helps to avoid complexity, it also alleviates some of the problems mentioned earlier. A component may be looked upon as a graph: the transistors (and the pads) are the vertices, and the connections the edges. Hierarchical components give rise to tree-like graphs. Balanced binary trees can be positioned in the plane very well (REM 79). The fact that we have tree-like graphs may, consequently, alleviate the layout problem. A tree is the graph that has the fewest edges while still being connected. Our confinement to tree-like graphs seems thus to result in circuits that satisfy VLSI's property of limited connectivity.

A component consists of subcomponents and connections between them. A connection equates the output of one component with the input of another. Therefore, we shall have to take input and output events into account. Actually, we shall phrase the meaning (the semantics) of a component as a relation between input and output events. Input and output events are partially ordered. Events that are not ordered are called concurrent. We do not postulate that certain occurrences of events must overlap in time. We take the complementary view: certain occurrences must be ordered. The more partial the order - i.e., the more concurrency - the more freedom we leave

to the implementation to have the occurrences of events overlap in time. For the formulation of the meaning of a component we shall, rather than the partial order itself, use the set of all sequences of input and output events that comply with the partial order. Such a sequence of events is called a trace. The meaning of a component is a set of traces, each trace being a finite-length sequence of events. This approach bears resemblance to path expressions (CAMPBELL & HABERMANN 74). It differs from COSY (LAUER 81) in that we use traces rather than vectors of traces. The reason for our preference is that vectors of traces exhibit the internal structure of components and thus do not lend themselves well to hierarchical composition. Trace theory is the subject of Section 2.

How do we cope with delays in connections? The traditional method is to ignore them. But, as we said earlier, that cannot be done in VLSI (SEITZ 79). We could try to estimate the delay times, but that would make the correct functioning of our components dependent on the way in which the connections are placed in the silicon chip. We take the position that delay times are unbounded (STUCKI & COX 79), and that the components should function correctly irrespective of the delay times. Such components are called delay-insensitive (or self-timed). They are discussed in Section 4. Making our components delay-insensitive will also alleviate the layout problem, since it does not impose upper bounds on the lengths of connections.

2. TRACE THEORY

In this section we discuss the concept of a trace structure. We introduce composition rules and we show how trace structures can be used to define the semantics of partially ordered computations.

2.1. Trace structures and composition rules

Let Σ be an infinite set of symbols. A *trace structure* is a pair $<T,A>$, in which A is a finite subset of Σ, and $T \subset A^*$. A^* denotes, as usual, the set of all finite-length sequences of elements of A, including the empty sequence which is denoted by ε. A is called the *alphabet* of the trace structure, and T its *trace set*. The elements of T are called *traces*.

Let $S = <T,A>$ and $S' = <T',A'>$ be trace structures, and let h be a function $h: A^* \to (A')^*$ such that

(i) $h(\varepsilon) = \varepsilon$;

(ii) $h(ab) = h(a)h(b)$;

then h is a homomorphism from S to S'. (Concatenation is denoted by juxtaposition.) The homomorphism π_B given by $\pi_B(a) = a$ for $a \in B$ and $\pi_B(a) = \varepsilon$ for $a \notin B$ is called the *projection* on alphabet B. In words, $\pi_B(t)$ is trace t from which all symbols not in B have been deleted. The projection of a trace set T on alphabet B is the trace set

$$\{\pi_B(t) \mid t \in T\}.$$

It will be denoted by $\pi_B(T)$. The projection of a trace structure $S = \langle T,A \rangle$ on alphabet B is the trace structure

$$\langle \pi_B(T),\ A \cap B \rangle.$$

We denote it by $\pi_B(S)$.

PROPERTY 2.1. $\pi_A \circ \pi_B = \pi_{A \cap B}$

PROPERTY 2.2. Let $s \in A^*$ and $t \in B^*$ such that $\pi_B(s) = \pi_A(t)$. Then

$$(\exists\ u \in (A \cup B)^*\colon \pi_A(u) = s \wedge \pi_B(u) = t).$$

PROOF. A trace u can be constructed as follows. If s or t is ε, take as u the other one. If s and t both start with a symbol in $A \cap B$ they start with the same symbol. Take that as the first symbol of u. Otherwise, take a first symbol of s or t that is not in $A \cap B$. Construct the remainder of u in the same way out of the remainders of s and t. □

The *p-composition* of two trace structures $S = \langle T,A \rangle$ and $S' = \langle U,B \rangle$, denoted $S\ \underline{p}\ S'$, is the trace structure

$$\langle \{t \in (A \cup B)^* \mid \pi_A(t) \in T \wedge \pi_B(t) \in U\},\ A \cup B \rangle.$$

Whenever obvious from the context, the alphabets may be deleted from the trace structures and we simply talk of the p-composition of trace sets.

EXAMPLE 2.1. (The alphabets associated with the following trace sets are

assumed to be chosen as small as possible.)

$$\{ab, cd\} \ \underline{p} \ \{be, df\} = \{abe, cdf\}$$

$$(\{ab\} \underline{p} \{ac\}) \ \underline{p} \ \{ac\} = \{abc, acb\} \ \underline{p} \ \{ac\} = \{abc, acb\}$$

$$\{ab\} \ \underline{p} \ (\{ac\} \underline{p} \{ac\}) = \{ab\} \ \underline{p} \ \{ac\} = \{abc, acb\}$$

PROPERTY 2.3. P-composition is idempotent, symmetric, and associative.

As an aside, we mention that p-composition can also be defined with inverse homomorphisms. The inverse of π_B, π_B^{-1}, is defined as follows. Let T be a trace set. Then

$$\pi_B^{-1}(T) = \{u \in \Sigma^* \mid \pi_B(u) \in T\}.$$

It satisfies

$$\pi_B(\pi_B^{-1}(T)) = T \text{ for } T \subset B^*.$$

The p-composition of S = <T,A> and S' = <U,B> is the trace structure

$$<\pi_{A \cup B}(\pi_A^{-1}(T) \cap \pi_B^{-1}(U)), A \cup B>.$$

Since regular sets are closed under (inverse) homomorphism and intersection (HOPCROFT & ULLMAN 69), we have the following property.

PROPERTY 2.4. If T and U are regular sets then $T \ \underline{p} \ U$ is a regular set.

PROPERTY 2.5. Let <T,A> be a trace structure. Then

$$\pi_A(T \underline{p} T') \subset T \quad \text{for all } T'.$$

PROOF. Let $u \in \pi_A(T \underline{p} T')$. Let $t \in T \ \underline{p} \ T'$ such that $u = \pi_A(t)$. Since $t \in T \ \underline{p} \ T'$, we have $\pi_A(t) \in T$ and, hence, $u \in T$. □

Property 2.5 with the inclusion replaced by equality will hold only for special choices of T and T'. This is reflected in the following definition. Two trace structures S = <T,A> and S' = <U,B> are said to *match* when $\pi_B(T) = \pi_A(U)$.

PROPERTY 2.6. Two trace structures S = <T,A> and S' = <U,B> match if and only if

$$(2.1)\qquad \pi_A(T\ \underline{p}\ U) = T \wedge \pi_B(T\ \underline{p}\ U) = U.$$

PROOF. (i) Assume S and S' match. We prove $T \subset \pi_A(T\ \underline{p}\ U)$. Equality then follows from Property 2.5. For reasons of symmetry, we then also have $\pi_B(T\ \underline{p}\ U) = U$.
Let $t \in T$. Let $u \in U$ be such that $\pi_B(t) = \pi_A(u)$. The fact that S and S' match implies that such a u exists. Let $s \in (A \cup B)^*$ be such that $\pi_A(s) = t \wedge \pi_B(s) = u$. From Property 2.2 we know that such an s exists. Then $s \in T\ \underline{p}\ U$, which, combined with $t = \pi_A(s)$, yields $t \in \pi_A(T\ \underline{p}\ U)$.
(ii) Assume (2.1).

$$\begin{aligned}\pi_A(U) &= \pi_A(\pi_B(T\ \underline{p}\ U)) \\ &= \pi_{A\cap B}(T\ \underline{p}\ U) \\ &= \pi_B(\pi_A(T\ \underline{p}\ U)) \\ &= \pi_B(T)\quad . \qquad \square\end{aligned}$$

PROPERTY 2.7. The property of matching is reflexive and symmetric, but not transitive.

EXAMPLE 2.2.

$$\begin{aligned}S_0 &= \langle\{abc,\ de\},\ \{a,b,c,d,e\}\rangle \\ S_1 &= \langle\{bcf,\ dg\},\ \{b,c,d,f,g\}\rangle \\ S_2 &= \langle\{cbf,\ dg\},\ \{b,c,d,f,g\}\rangle \\ S_3 &= \langle\{fg\ \},\ \{f,g\}\rangle \\ S_4 &= \langle\{bcf,\varepsilon\},\ \{b,c,d,f,g\}\rangle \\ S_5 &= \langle\{bcf,\varepsilon\},\ \{b,c,f\}\rangle \\ S_6 &= \langle\{bcf\},\ \{b,c,f\}\rangle\ .\end{aligned}$$

Trace structure S_0 matches S_1, S_3, and S_5. It does not match the other three trace structures.

Let $S = \langle T,A\rangle$ be a trace structure. We call $s \in A^*$ a *prefix* of trace $t \in T$ if $(\exists u \in A^*: t = su)$. The set of all prefixes of traces in T is denoted by PREF(T). A trace set T satisfying PREF(T) = T is called *prefix-closed*.

Let a relation $\sim$ be defined on PREF(T) by $s \sim t$ if and only if

$$\{u \in A^* \mid su \in T\} = \{u \in A^* \mid tu \in T\}.$$

Relation $\sim$ is a right congruence and, hence, an equivalence relation. We call the equivalence classes of $\sim$ the *states* of S. We denote the equivalence class (state) of which s is a member by $[s]_S$. Whenever S is obvious from the context, it is omitted. For regular T the relation $\sim$ is of finite index (HOPCROFT & ULLMAN 69), or, phrased differently, if T is a regular set the number of states is finite.

Trace structures are used for defining the semantics of partially ordered computations. The trace structure of a computation is the composition of the trace structures of the subcomputations. The trace structure of the composite should not reflect the internal relations of the subcomputations. Therefore, we introduce a second composition operation, q-composition, which is the p-composition followed by the elimination of all common symbols.

The *q-composition* of two trace structures $S = \langle T,A\rangle$ and $S' = \langle U,B\rangle$, notation $S \; \underline{q} \; S'$, is the trace structure

$$\langle \pi_{A \div B}(T \; \underline{p} \; U),\; A \div B\rangle$$

($\div$ denotes symmetric set difference, i.e. $A \div B = (A \cup B) \setminus (A \cap B)$.) This composition operator differs from the one in MILNER 80 in that the latter one replaces common symbols by "silent moves" τ.

PROPERTY 2.8. Q-composition is symmetric.

EXAMPLE 2.3. (cf. Example 2.1)

$$\{ab, cd\} \; \underline{q} \; \{be, de\} = \{ae, cf\}$$

$$(\{ab\} \; \underline{q} \; \{ac\}) \; \underline{q} \; \{ac\} = \{bc, cb\} \; \underline{q} \; \{ac\} = \{ab, ba\}$$

$$\{ab\} \; \underline{q} \; (\{ac\} \; \underline{q} \; \{ac\}) = \{ab\} \; \underline{q} \; \{\varepsilon\} \qquad = \{ab\}.$$

The example above shows that q-composition is not associative. In order to

achieve associativity, we must have, for trace structures $\langle T,A\rangle$, $\langle U,B\rangle$, and $\langle V,C\rangle$,

$$\pi_{A \div B \div C}(\pi_{A \div B}(T \, \underline{p} \, U) \, \underline{p} \, V) = \pi_{A \div B \div C}(T \, \underline{p} \, U \, \underline{p} \, V).$$

This is true if the symbols deleted by $\pi_{A \div B}$ do not occur in the traces in V, i.e. if $A \cap B \cap C = \emptyset$. This is expressed by the following Property.

PROPERTY 2.9. For trace structures $S = \langle T,A\rangle$, $S' = \langle U,B\rangle$, and $S'' = \langle V,C\rangle$ such that $A \cap B \cap C = \emptyset$ we have

$$(S \, \underline{q} \, S') \, \underline{q} \, S'' = S \, \underline{q} (S' \, \underline{q} \, S'').$$

PROPERTY 2.10. The p-composition and the q-composition of prefix-closed trace sets are prefix-closed.

PROPERTY 2.11. Let $S = \langle T,A\rangle$ and $S' = \langle U,B\rangle$ be two matching trace structures. Then

$$\mathrm{PREF}(T \, \underline{q} \, U) = \mathrm{PREF}(T) \, \underline{q} \, \mathrm{PREF}(U).$$

There is a special trace structure called SYNC. Let k be a natural number and let a and b be two distinct symbols. $\mathrm{SYNC}_k(a,b)$ is the trace structure with $\{a,b\}$ as its alphabet and

$$\{t \in \{a,b\}^* \mid 0 \le \#_a t' - \#_b t' \le k \text{ for every prefix } t' \text{ of } t\}$$

as its trace set. ($\#_a t'$ denotes the number of occurrences of a in t'.) $\mathrm{SYNC}_k(a,b)$ is, consequently, prefix-closed. It has k + 1 states, viz. the elements of the set

$$\{[a^i] \mid 0 \le i \le k\}$$

$(a^0 = \varepsilon;\ a^{i+1} = a^i a)$.

PROPERTY 2.12. Let a, b, and c be three distinct symbols. Then

$$\mathrm{SYNC}_k(a,b) \, \underline{q} \, \mathrm{SYNC}_m(b,c) = \mathrm{SYNC}_{k+m}(a,c).$$

A number of proofs that have been omitted in these notes may be found in VAN DE SNEPSCHEUT 82.

2.2. Programs denoting partially ordered computations

We introduce a program notation. Every program denotes a partially ordered computation and thus defines a trace structure. A program is a hierarchy of commands: a program is a component, every component consists of a command and a number of subcomponents with relations between their alphabets. First we define the trace set TR(S) defined by a command S. Since a command has one of five possible forms, we define TR inductively by five cases. (b denotes a symbol; S_0, S_1, and S denote commands.)

(i) A symbol is a command

$$TR(b) = \{b\}$$

(ii) $S_0 \mid S_1$ is a command

$$TR(S_0 | S_1) = TR(S_0) \cup TR(S_1)$$

(iii) S_0, S_1 is a command

$$TR(S_0,S_1) = TR(S_0) \, \underline{q} \, TR(S_1)$$

(iv) $S_0 \, ; S_1$ is a command

$$TR(S_0;S_1) = \{t_0 t_1 \mid t_0 \in TR(S_0) \wedge t_1 \in TR(S_1)\}$$

(v) S^* is a command

$$TR(S^*) = TR(S)^*.$$

Except for (iii), the trace structure of a command has as its alphabet the union of the alphabets of its parts. Where appropriate, TR(S) denotes the trace structure of S. Notice that trace sets of commands are not necessarily profix-closed.

If $TR(S_0)$ and $TR(S_1)$ are trace sets with disjoint aphabets we have

$$TR(S_0,S_1) = TR(S_0)\ \underline{q}\ TR(S_1)$$
$$= TR(S_0)\ \underline{p}\ TR(S_1).$$

In this case, both compositions amount to the shuffle (GINSBURG 66) operation. Since regular sets are closed under q-composition, TR(S), for S a command, is a regular set.

In order to save on parentheses, we introduce the rule that the comma has the highest priority, followed by the semicolon, and then the vertical bar:

$$S_0,S_1 \mid S_2 = (S_0,S_1) \mid S_2$$
$$S_0,S_1;S_2 = (S_0,S_1);S_2$$
$$S_0;S_1 \mid S_2 = (S_0;S_1) \mid S_2$$

PROPERTY 2.13.

(i) $TR(S_0|S_1) = TR(S_1|S_0)$

(ii) $TR(S_0,S_1) = TR(S_1,S_0)$

(iii) $TR((S_0|S_1)|S_2) = TR(S_0|(S_1|S_2))$

(iv) $TR((S_0;S_1);S_2) = TR(S_0;(S_1;S_2))$

(v) $TR((S_0,S_1),S_2) = TR(S_0,(S_1,S_2))$ provided $\bigcap_{i=0}^{2}(\text{alphabet of } S_i) = \emptyset$

(vi) $TR(S_0;(S_1|S_2)) = TR(S_0;S_1|S_0;S_2)$

(vii) $TR((S_0|S_1);S_2) = TR(S_0;S_2|S_1;S_2)$

(viii) $TR(S_0,(S_1|S_2)) = TR(S_0,S_1|S_0,S_2)$

(ix) $TR(S^{**}) = TR(S^*)$

(x) $TR(S^*;S^*) = TR(S^*)$.

Property 2.13(vi) does not hold in MILNER 80; there the commands $S_0;(S_1|S_2)$ and $S_0;S_1|S_0;S_2$ are considered to be different.

A program is a component. The simplest form a component can have is a single command. Syntactically, such a component is of the form

$\underline{\text{com}}$ C("alphabet of S"): S $\underline{\text{moc}}$

C is the name of the component. Its trace structure is

<PREF(TR(S)), alphabet of S> .

Thus, components have a prefix-closed trace set. In HOARE 78 PREF is applied to trace sets of commands. This requires the introduction of a termination symbol "$\surd$" to cater for sequential composition (the semicolon). A termination symbol at the end of a trace indicates that it is not a trace brought about by PREF. Since we do not have sequential composition for components, there is no need to introduce a termination symbol.

The following is an example of a component.

com binsem(v,p): $(v;p)^*$ moc.

Its trace structure is $SYNC_1(v,p)$.

We now turn to components that have subcomponents as well as a command. Each subcomponent has a name and a type. The type of a subcomponent is a component. To differentiate between the alphabets of the subcomponents we introduce composite symbols. A composite symbol is a symbol s.a, in which s is a name of a subcomponent and a a symbol in the alphabet of that subcomponent's type. Symbols that are not composite are called simple. The alphabet of a component contains simple symbols only.

Let s be the name of a subcomponent of type C. Let C have <T,A> as its trace structure. The trace structure of s is then

<s.T, s.A>

in which s.T is the trace set obtained from T by replacing in each trace of T every symbol a by the composite symbol s.a. The alphabet s.A is likewise obtained from A by changing each symbol a into s.a.

Syntactically, a component with subcomponents is of the form

com C(A):

  sub s_0: C_0, ..., s_{n-1}: C_{n-1}

  S

moc

A is an alphabet of simple symbols. Component C has n subcomponents s_i $(0 \le i < n)$. The n names s_i must be distinct. The type of s_i is C_i. Let C_i have alphabet A_i, then S is a command with

$$A \cup \bigcup_{i=0}^{n-1} s_i.A_i$$

as its alphabet. The trace set of C is given by

$$\text{(2.2)} \qquad TR(C) = PREF(TR(S)) \ \underline{q}\ s_0.TR(C_0)\ \underline{q}\ \dots\ \underline{q}\ s_{n-1}.TR(C_{n-1}).$$

Since the alphabets of the subcomponents are subsets of the alphabet of S, the alphabet of C is A and thus contains simple symbols only. Notice also that TR(C) is prefix-closed. Due to the prefixing with the subcomponent's name, the alphabets of the subcomponents are disjoint. Property 2.9 then ensures the associativity of the q-composition in (2.2).

In these notes we want to restrict ourselves to regular trace sets. Therefore, we do not allow components to be recursive. More precisely: we say that component C has component D as a composing part if C has a subcomponent that either is of type D or has D as a composing part. We do not allow components that have themselves as a composing part. This restriction makes (2.2) a nonrecursive equation.

The following is an example of a component with subcomponents.

```
com sexsem(v,p):
    sub b0,b1: binsem
    (v; b0.v)*, (b0.p; b1.v)*, (b1.p; p)*
moc
```

The second line is short for "sub b0: binsem, b1: binsem". According to Properties 2.11 and 2.12, the trace structure of sexsem is $SYNC_5(v,p)$.

By extending the alphabet of binsem with a third symbol we are able to achieve a stronger synchronization between the two subcomponents:

```
com binsem'(v,p,q): (v;p;q)* moc
com quinsem(v,p):
    sub b0,b1: binsem'
    (v; b0.v)*, (b0.p; b1.v; b0.q; b1.q)*, (b1.p; p)*
moc
```

Consider the p-composition of the trace sets of b0, b1, and the middle part of the command of quinsem:

(2.3) $(b0.v;\ b0.p;\ b0.q)^*$

(2.4) $(b1.v;\ b1.p;\ b1.q)^*$

(2.5) $(b0.p;\ b1.v;\ b0.q;\ b1.q)^*$

In each trace t of that p-composition the (i+2)nd b0.v follows, because of (2.3), the (i+1)st b0.q and hence, because of (2.5), the (i+1)st b1.v and hence, because of (2.4), the i-th b1.p. Hence

$$\#_{b0.v}\, t - \#_{b1.p}\, t \le 2 \ .$$

The q-composition of the three trace sets is $SYNC_2$(b0.v, b1.p). By dropping in (2.3) through (2.5) the symbols b0.q and b1.q, the q-composition would, according to Property 2.11, yield $SYNC_3$(b0.v, b1.p). Thus we have achieved a stronger synchronization between the two subcomponents. By q-composing $SYNC_2$(b0.v, b1.p) with the trace sets of the other two parts of the command of quinsem we get, again using Properties 2.11 and 2.12, $SYNC_4$(v,p) as the trace structure of quinsem.

We introduce a simpler mechanism to achieve stronger synchronization between (sub)components. Symbols a and b in different alphabets may be equated by adding an equation "a = b" to the component definition. The following component contains an example of this.

```
com quinsem'(v,p):
    sub b0,b1: binsem
    b0.p = b1.v
    (v; b0.v)*, (b1.p; p)*
moc
```

In quinsem' the alphabets of b0 and b1 are not disjoint. Their intersection contains one symbol.

We have thus arrived at the most general form a component can have

com C(A):

  sub s_0: C_0, ..., s_{n-1}: C_{n-1}

  $a_0 = b_0$, ..., $a_{m-1} = b_{m-1}$

  S

moc

Let C_i have alphabet A_i. The symbols occurring in the equations must be symbols of the alphabets A, $s_0.A_0$, ..., $s_{n-1}.A_{n-1}$. Two symbols occurring in the same equation must be chosen from two different alphabets. The trace set of C is again given by (2.2). Let B be the set of all symbols that occur in an equation. Each symbol in B must occur exactly once in the equations. The alphabet of S is $(A \cup \bigcup_{i=0}^{n-1} s_i.A_i) \backslash B$. Thus we achieve again that the q-composition in (2.2) is associative. We allow the command S to be empty. In that case, TR(S) in equation (2.2) is by definition equal to $\{\varepsilon\}$. Having defined the trace structure of a component with equations, we can use Properties 2.11 and 2.12 again to show that quinsem' has $SYNC_4(v,p)$ as its trace structure.

We now have two ways of expressing communication between (sub)components: by introducing an equation between symbols in their alphabets or by having the symbols occur in the command of the component. The former way is equivalent to the way communication is expressed in HOARE 78a. In MARTIN 81 the concept of "synchronization slack" is introduced. The slack is the number of steps two synchronized processes are allowed to be out of step. A slack = 0 between symbols a and b corresponds to an equation a = b in our

formalism. A slack = k for k > 0 between a and b corresponds to a q-composition with the trace structure $SYNC_k(a,b)$.

The following is an example of a component in which all symbols in the alphabets of the component and the subcomponents occur in the equations.

```
com trisem(v,p):
    sub b0,b1: binsem
    v = b0.v, b0.p = b1.v, b1.p = p
moc
```

Its trace structure is, again according to Properties 2.11 and 2.12, $SYNC_2(v,p)$.

2.3. Examples of components

EXAMPLE 2.4.

$$\underline{\text{com}}\ \text{buf1}(x0,x1,y0,y1):\ (x0;\ y0 \mid x1;\ y1)^*\ \underline{\text{moc}}$$

This component has three states: [ε], [x0], and [x1]. It may be interpreted as a one-bit buffer. The symbols x0 and x1 then stand for "receive a 0" and "deliver a 1", respectively. Using this interpretation, the three states stand for "buffer empty", "buffer contains a 0", and "buffer contains a 1".

EXAMPLE 2.5.

$$\underline{\text{com}}\ \text{boolvar}(x0,x1,y0,y1):\ (x0;\ y0^* \mid x1;\ y1^*)^*\ \underline{\text{moc}}$$

This component has the same three states as buf1. However, in this case the state [ε] contains the trace ε only, and thus stands for "component uninitialized". The component may be interpreted as a boolean variable. Notice that the component is constructed in such a way that is must be initialized before it can inspected.

EXAMPLE 2.6.

```
com queue_1(x0,x1,y0,y1) : (x0; y0 | x1; y1)* moc
```

for i > 1 :

```
com queue_i(x0,x1,y0,y1):
    sub q: queue_{i-1}
    q.y0 = y0, q.y1 = y1
    (x0; q.x0 | x1; q.x1)*
moc
```

Component queue_i ($i \geq 1$) is an i-bit buffer. For $i = 1$ it is equal to component buf1. For $i > 1$ it consists of a one-bit buffer between its inputs and the inputs to its subcomponent, as expressed by its command, and an (i-1)-bit buffer as a subcomponent. Notice that the component does not violate our restriction on recursion: queue_i does not have queue_i as a composing part.

We can replace the command of queue_1 by a suitable subcomponent and add the appropriate equations

```
com queue_1(x0,x1,y0,y1):
    sub b: buf1
    x0 = b.x0, x1 = b.x1, b.y0 = y0, b.y1 = y1
moc
```

We can change queue_i in a similar fashion ($i>1$)

```
com queue_i(x0,x1,y0,y1):
    sub b: buf1, q: queue_{i-1}
    x0 = b.x0, x1 = b.x1, b.y0 = q.x0, b.y1 = q.x1,
    q.y0 = y0, q.y1 = y1
moc
```

Thus we can, if we wish, change any component into one that has either no subcomponents (and, consequently, no equations) or no command.

EXAMPLE 2.7. This example is a binary stack of depth i. A binary stack has an alphabet of four symbols: two opening parentheses, $(_0$ and $(_1$, and two closing parentheses, $)_0$ and $)_1$. Consider all well-nested sequences of these parenthese. For example,

$$(_0\ (_1\ )_1\ (_0\ (_1\ )_1\ )_0\ )_0\ (_1\ )_1$$

is well-nested, but

$$(_0\ )_1\ (_1\ )_0 \quad \text{and} \quad (_0\ (_1\ )_0\ )_1$$

are not. The trace set of a stack is the set of all prefixes of well-nested sequences. Our example will not be a general stack but a stack of depth i. Consider a prefix of a well-nested sequence. Its nesting level is defined as the number of opening parentheses minus the number of closing parentheses, and its nesting depth as the maximum of the nesting levels of its prefixes. The trace set of a stack of depth i is the set of all prefixes of well-nested sequences with nesting depth $\le$ i.

In the component the opening parentheses are denoted by x0 and x1, and the closing parentheses by y0 and y1.

$$\underline{\text{com}}\ \text{stack}_1(\text{x0,x1,y0,y1}):\ (\text{x0, y0} \mid \text{x1; y1})^*\ \underline{\text{moc}}$$

for i > 1:

```
com stack_i(x0,x1,y0,y1):
    sub s: stack_{i-1}
    ( (x0; s.x0 | x1; s.x1)*;
      (x0; y0 | x1; y1);
      (s.y0; y0 | s.y1; y1)* )*
moc
```

Component $stack_1$ is obviously a binary stack of depth 1. Now consider $stack_i$ for $i > 1$. Assume that subcomponent s is a binary stack of depth $i - 1$. Leave out the middle line of the command of $stack_i$. Then component $stack_i$ is, because of the exact matching of its symbols with the symbols of s, also a binary stack of depth $i - 1$. The effect of adding the middle line is that at every position in a trace where an opening parenthesis is immediately followed by a closing parenthesis one pair of matching parentheses is inserted between them. Hence, component $stack_i$ is a binary stack of depth i.

The program above defines a stack, but it also exhibits the usage of a stack, viz. the usage of subcomponent s. It is a complicated program. Its intricacy becomes apparent if one wonders how the component is "executed", i.e. how a trace is selected that matches the component's environment. Or more specifically: how is the number of steps for each repetition determined? Consider the repetition in the first line of the command of $stack_i$. Let S denote

```
(x0; s.x0 | x1; s.x1) ;
(x0; y0 | x1; y1) .
```

Then S has the same trace set as

```
x0; (y0 | s.x0; S)
|x1; (y1 | s.x1; S).
```

We see that every trace in TR(S) starts with x0 or x1. The traces in the environment contain the symbols x0, x1, y0, and y1. (Neglect any other symbols for this discussion.) So the environment determines whether the first or the second line above is chosen. Assume the first line is chosen. Then next a choice must be made between y0 and "s.x0; S". If the environment has a trace with y0 as its next symbol y0 may be chosen and the repetition terminates. If there is a trace with x0 or x1 as its next symbol, "s.x0; S" may be chosen and the repetition continues. We are stuck when all traces have y1 as the next symbol: the trace sets of the component and the environment do not match. In the program of $stack_i$ than cannot occur: if s.y1 is a possible next symbol then so is s.y0.

EXAMPLE 2.8.

```
com fulladder(a0,a1,b0,b1,c0,c1,d0,d1,s0,s1):
    (a0,b0; d0,(c0; s0 | c1; s1)
    |a1,b1; d1,(c0; s0 | c1; s1)
    |(a0,b1 | a1,b0); (c0; d0,s1 | c1; d1,s0) )*
moc
```

This component may be interpreted as a full-adder element (cf. p. 250 of SEITZ 80). The pairs (a0,a1) and (b0,b1) represent the two bits to be added, (c0,c1) represents the carry-in, (d0,d1) the carry-out, and (s0,s1) the sum. The first line of the command is known as "carry-kill" and the second line as "carry-generate". In both cases the carry-out may precede the carry-in. The third line is known as "carry-propagate". In that case, of course, the carry-out has to follow the carry-in.

3. THE VLSI MEDIUM

It is our intention to realize the components defined in Section 2 as VLSI circuits. To gain insight in the problems associated with the translation of components into VLSI circuits we discuss the relevant properties of the VLSI medium. We consider in particular the realization of restoring logic circuitry in CMOS. An elementary introduction to integrated circuits may be found in CLARK 80.

3.1. CMOS

A Metal-Oxide-Semiconductor chip (MOS chip) is a thin layer of "substrate" with on top of it a network of conducting paths. A chip measures about 5 by 5 mm. The conducting paths are situated in a few layers. The different layers are separated from each other by an insulating material (siliconoxide). There are cuts through the oxide for the connection of paths in different layers. Proceeding from top to bottom we encounter one or two layers of metal (usually aluminum), a layer of poly-crystallized silicon, and one or two layers of doped silicon. The doped silicon is often called diffusion, after the way in which it is fabricated. Although in CMOS these layers are not made by diffusion, we shall adhere to that name.

In CMOS (Complementary CMOS) there are two types of diffusion, depending on the valence of the ions with which the silicon is doped. By doping silicon with ions we get a material in which either negative or positive charge carriers are abundant ("floating around"). The former material is called N-type diffusion, the latter P-type. They are both conductors. In N-type diffusion the charge carriers are electrons, in P-type holes (absent electrons). The substrate can be monolithic crystalline silicon or an insulator, such as sapphire. The latter choice has become known as CMOS/SOS (Silicon-On-Sapphire). SOS is a promising technology, since its insulating behaviour makes its physical properties very simple. Our discussion is mainly based on CMOS on an insulating substrate.

Wherever a polysilicon path crosses a diffusion path, a transistor is created: the voltage on the polysilicon path controls the flow of current through the diffusion path. These transistors are known as Field-Effect-Transistors or FETs. The polysilicon path is called the gate of the transistor. We shall call the diffusion path simply the path of the transistor. The part of the path that is underneath the gate is called the channel. Depending on the type of the path we distinguish N-type and P-type transistors. The voltage on the gate (V_g) determines the number of charge carriers in the channel. Increasing V_g attracts negative charge carriers, a decrease attracts positive charge carriers. Suppose there is a voltage difference between the two ends of the path. (Otherwise no current will flow through the path.) Phrased differently, suppose there are more charge carriers at one end of the path than at the other. We call the end with the excess of charge carriers the source and the other end the drain. (Notice that the distinction between source and drain is a dynamic one.) A transistor is on, i.e. its path is conveying charge carriers, when its gate attracts sufficiently many charge carriers from the source into the channel from where they flow to the drain. It is thus the voltage difference between V_g and V_s (source voltage) that determines whether a transistor is on. There is a threshold voltage V_t that determines the value of $V_g - V_s$ for which the transistor switches between on and off.

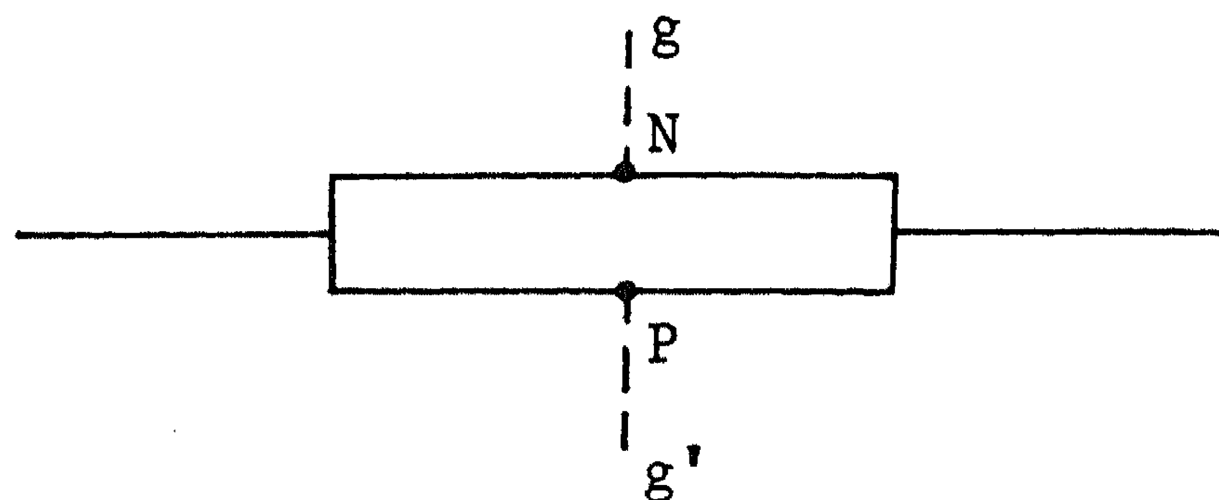
In an N-type transistor the charge carriers are electrons. Electrons are attracted by a high value of $V_g - V_s$ and repelled by a low value. The opposite is true for a P-type transistor:

		on when	off when
(3.1)	N-type	$V_g - V_s > V_t$	$V_g - V_s < V_t$
	P-type	$V_g - V_s < V_t$	$V_g - V_s > V_t$

By doping the channel with N-type or P-type impurities we can influence the V_t of the transistor. This technique is known as ion implantation. We call a transistor an enhancement transistor when it is "normally off", i.e. when it is off for $V_g = V_s$. Otherwise it is called a depletion transistor. In CMOS we do not have depletion transistors. According to (3.1), we could also have defined an enhancement transistor as a transistor of which the threshold voltage and the charge of the charge carriers have opposite signs.

Let 1 denote the positive voltage of the power supply, usually 3 to 5 V, and 0 the ground voltage. Now consider an N-type transistor. It is on for voltages V_s satisfying $0 \leq V_s < V_g - V_t$. For $V_s > V_g - V_t$ it is off and the drain voltage V_d also satisfies $V_d > V_g - V_t$, since otherwise the drain would be the source. For V_g maximal, i.e. $V_g = 1$, we find that an N-type transistor conveys only values V_s from source to drain that satisfy $0 \leq V_s < 1 - V_t$. It is a good conveyor for zeroes, but it corrupts ones. If such a corrupted one, say $1 - V_t$, is applied to the gate of a next transistor that one will convey only values V_s satisfying $0 \leq V_s < 1 - 2V_t$. An N-type transistor is a good switch only for zeroes, and that is the reason why we need a second type of transistor: the P-type transistor. In an analogous fashion we find that the P-type transistor is a good switch for ones but that it corrupts zeroes.

When designing VLSI circuits we want to abstract from the imperfection of the transistors. We want to compose VLSI circuits out of ideal switches and connections between them. This we can do in the following way. If a switch has to convey only ones or only zeroes we choose the appropriate transistor. If a switch has to convey both ones and zeroes we construct the switch out of an N-type and a P-type transistor



We use this combination only with complementary values, 0 and 1, on their gates. For g = 0 (and, hence, g'=1) both transistors are off. For g = 1 they are both on. If in the latter case a 0 is applied at one of the two sides the N-type transistor will convey it: the 0 is not corrupted. Similarly, a value 1 is conveyed perfectly by the P-type transistor.

In Section 3.2 we give some examples of VLSI circuits expressed in terms of switches and connections.

Another well-known MOS technology is NMOS. This is the technology discussed in the excellent introduction to VLSI MEAD & CONWAY 80. NMOS has only N-type transistors: enhancement and depletion transistors. The depletion transistors are used as resistors. Since a resistor is always somewhat on, NMOS has a higher power consumption and dissipation than CMOS. The asymmetry between the two types of transistors in NMOS has a number of annoying consequences, such as the need for pre-charging and ratio logic. In CMOS the only asymmetry between the types of transistors lies in the relative speeds of the charge carriers: the speed of an electron is between two and three times that of a hole. If the threshold voltage V_t is chosen too small we get a phenomenon known as subthreshold leakage. This is more serious in NMOS than in CMOS, since the way in which transistors are used in CMOS allows V_t to be chosen larger than would be desirable for NMOS.

Over the years integrated circuits have become smaller in size. It is interesting to observe how a circuit's behaviour is affected when all its dimensions are scaled down. In integrated circuits time is usually measured in multiples of τ, the transition time of transistor. It satisfies

$$\tau = \frac{L^2}{\mu(V_d - V_s)}$$

in which L is the distance from source to drain (the channel length) and μ the mobility of the charge carriers in the channel. We reduce the spatial dimensions, including those vertical to the substrate, by multiplying them by a factor α ($0<\alpha<1$). Thus $L' = \alpha L$. In order to keep the electric field in the channel constant, we multiply the voltage by α as well. This results in $\tau' = \alpha\tau$: the transit time is reduced by the same factor. Another consequence of the fact that we scale the voltage down is that the power dissipation per unit area remains the same.

The time required for a signal to travel through a path from one transistor to another is proportional to the product of the resistance and the

capacitance of the path. The resistance R of a path is proportional to its length and inversely proportional to its cross section. Hence $R' = R/\alpha$. The capacitance of a path is inversely proportional to the distance to its neighbouring paths and layers, and it is proportional to the area facing that neighbouring path or layer. Hence, $C' = \alpha C$, and, consequently, $R'C' = RC$. The time required for a signal to go from one transistor to another measured in seconds is, consequently, not affected. But since $\tau' = \alpha\tau$, the time measured in multiples of τ has changed. In τ-relative terms, if it took a signal time t to go from one transistor to another then after scaling it will take time t/α. Thus the scaled circuit may not function anymore. The delays in its paths may have become too long.

Matters are even worse if we look at the time required for a signal to go a fixed distance, say from one end of the chip to the other. If in τ-relative terms it first took time t, then after scaling it will take time t/α^2. This clearly demonstrates, firstly why propagation delays cannot be ignored in VLSI, and secondly that the distribution of a clock signal over the entire chip is not viable in VLSI, since such a cross-chip propagation has a quadratic scaling factor. The reader will now appreciate why we are looking for delay-insensitive circuits.

3.2. Restoring logic circuitry

We have demonstrated how perfect switches can be realized in CMOS. We now use these switches to construct restoring logic circuitry. A restoring logic circuit is a circuit in which the outputs are permanently driven by the power supply. The inputs determine, by controlling the switch settings, which outputs are connected to the high voltage of the power supply, denoted by 1, and which ones to the ground voltage, denoted by 0. We do, consequently, not rely on the fact that values may be stored temporarily on disconnected paths. (Due to the subthreshold leakage, values on disconnected paths deteriorate with time.) Restoring logic seems the natural choice for circuits that are realized in a submicron technology i.e. with path widths smaller than 10^{-6} m.

A *logic component* is a graph. Its vertex set is the union of the disjoint sets X, Y, Z, and {0,1}. The elements of X and Y are called *input ports* and *output ports*, respectively. The elements of Z are called *interior nodes*. Each edge of the graph is either labeled or unlabeled. (Labeled edges represent switches.) There are two types of labels: i and i', with $i \in X$. A

label i represents a switch that is on for i = 1, i' a switch that is on for i = 0. Logic components will often simply be called components. (In hierarchically composed components, to be discussed below, we allow labels that are chosen from a larger set than just X.)

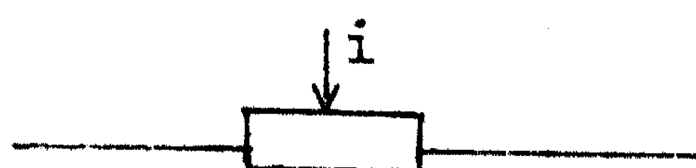
P(X) denotes the power set of X. Vertices j and k are called *separated* if for every $A \in P(X)$ every path between j and k has an edge labeled i with $i \notin A$ or an edge labeled i' with $i \in A$. Let $V \subset X \cup \{0,1\}$. Vertex j is called *driven by* V if for every $A \in P(X)$ there exists a $v \in V$ and a path between j and v for which $i \in A$ for every edge labeled i and $i \notin A$ for every edge labeled i'. A logic component is called *nonfighting* if every two distinct vertices in $X \cup \{0,1\}$ are separated. A logic component is called *well-behaved* if it is nonfighting and every vertex $j \in Y$ is driven by $X \cup \{0,1\}$. A logic component is called *restoring* if it is nonfighting and every vertex $j \in Y$ is driven by $\{0,1\}$. Notice that a restoring component is well-behaved.

We introduce a notation for the description of logic ocmponents that is very similar to the one we introduced for partially ordered computations. The difference is that the alphabet is replaced by a list of ports and the command by an enumeration of the edges of the graph. The following is an example of a description of a component.

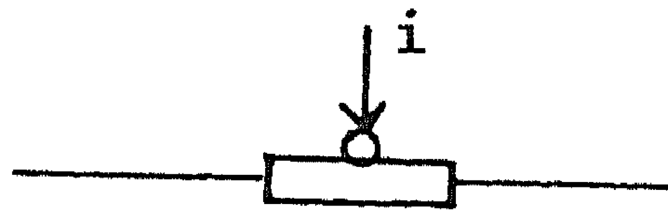
```
com inverter(in?,out!):
    in' → out = 1
    in → out = 0
moc
```

In the list of ports each input port is postfixed by a question mark and each output port by an exclamation point. The graph of this component has two edges: an edge labeled in', connecting out and 1, and an edge labeled in, connecting out and 0. It has no interior nodes. It is an example of a restoring logic component.

For every logic component we can draw a diagram. This is essentially a picture of the graph extended with connections between input ports and the switches they control. An edge labeled i is drawn as



and an edge labeled i' as



They represent the switches. Their gates are connected to input port i. Arrows with the same label may be drawn connected in the diagram. Fig. 1 shows a diagram of the inverter.

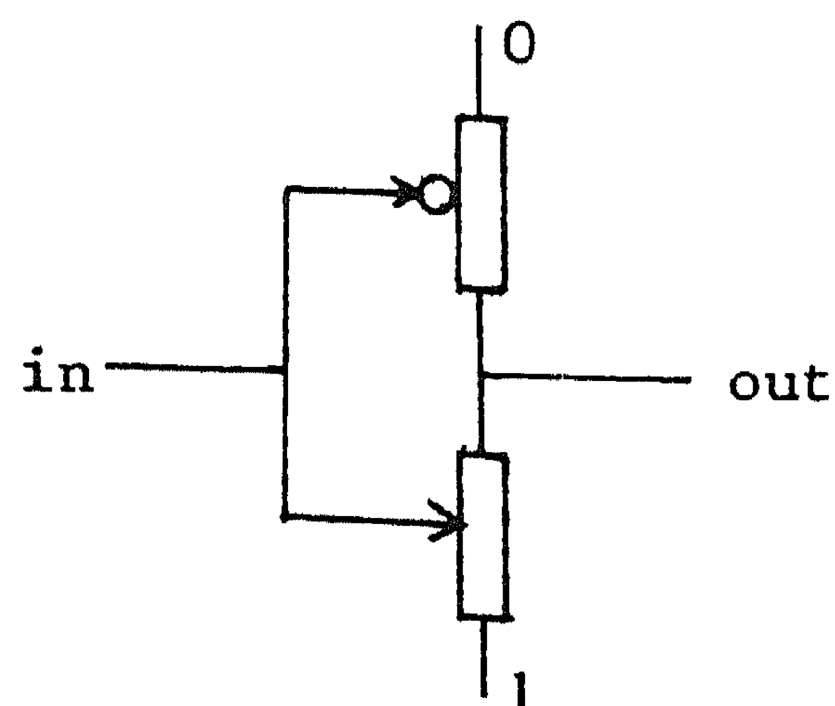


Fig. 1   Diagram of an inverter

When specifying labeled edges we allow more general Boolean expressions to the left of the arrow than just the labels i and i' ($i \in X$). Such labels may be connected by the Boolean operators $\wedge$ and $\vee$. The formula "$b_0 \vee b_1 \rightarrow x = y$" specifies two connections between x and y: $b_0 \rightarrow x = y$" and "$b_1 \rightarrow x = y$". This is known as parallel composition. The formula "$b_0 \wedge b_1 \rightarrow x = y$" introduces an interior node. Calling that interior node z, the formula is equivalent to "$b_0 \rightarrow x = z$" and "$b_1 \rightarrow z = y$". This is known as serial composition. Both parallel and serial composition occur in the following component.

```
com nor(a?,b?,out!):
    a' ∨ b' → out = 1
    a ∨ b → out = 0
moc
```

A diagram of this component is shown in Fig. 2.

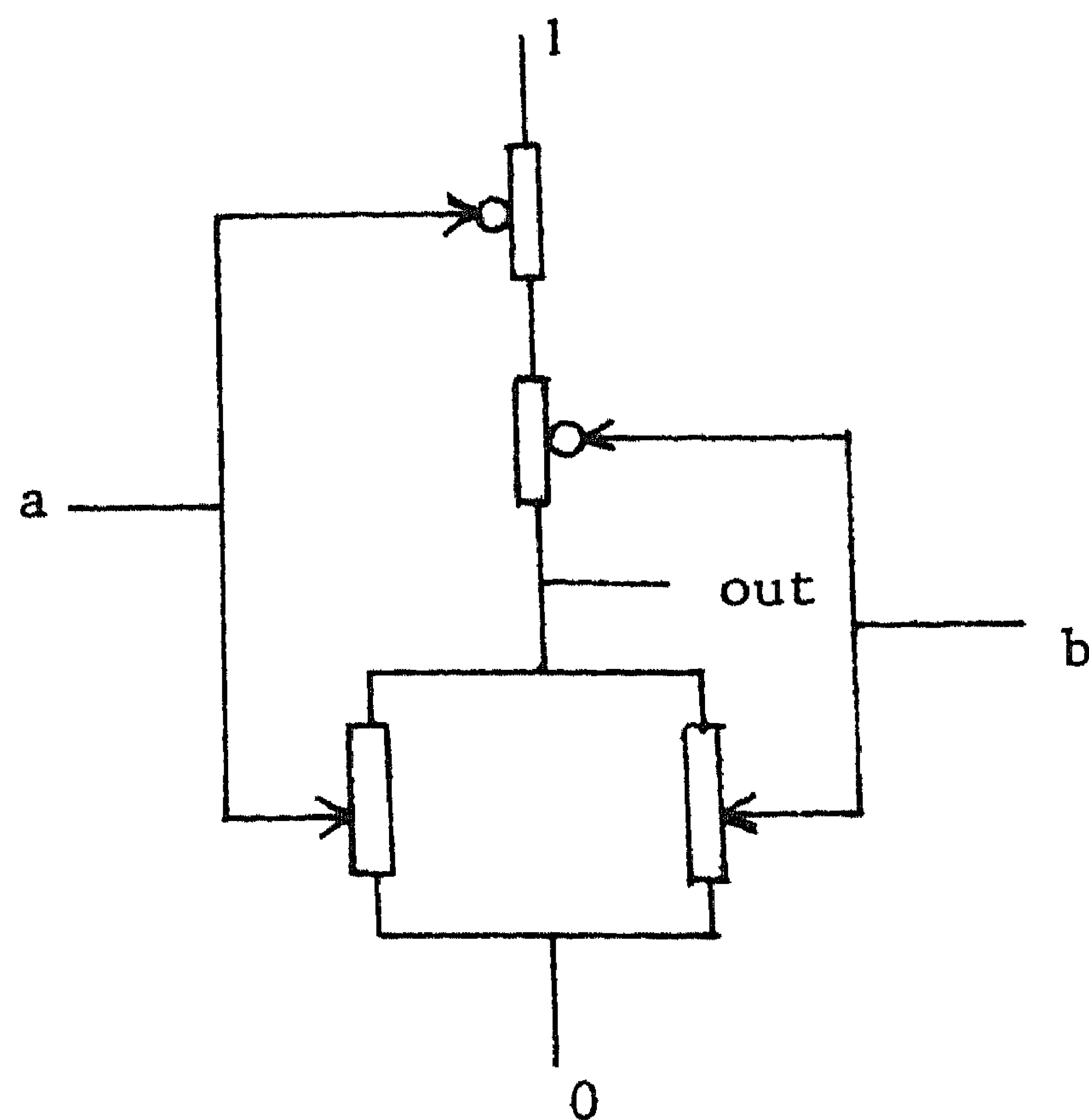


Fig. 2 Diagram of a nor-circuit

Except in very simple cases, components will be composed of subcomponents and connections between them. The latter are (possibly labeled) connections between the ports of the component, {0,1}, interior nodes, and the ports of the subcomponents. These connections thus constitute a logic component, to which we shall refer as the local graph.

A *hierarchically composed component* consists of a logic component, called the *local graph,* and zero or more hierarchically composed subcomponents. X_{loc} and Y_{loc} denote the sets of input and output ports, respectively, of the hierarchically composed component. X_{int}, called the set of *internal input ports* of the local graph, is the union of the sets of output(!) ports of the subcomponents. Y_{int}, the *internal output ports,* is the union of the sets of input(!) ports of the subcomponents. (For a component without subcomponents the local graph is, consequently, the entire component.)

A hierarchically composed component is a component. Its graph is the composition of the local graph and the graphs of the subcomponents, in which the ports of the subcomponents are the internal ports of the local graph. Let X_{tot} be defined as the union of X_{loc} of the local graph and the X_{tot}'s of the subcomponents. A hierarchically composed component has labels i and i' with $i \in X_{tot}$. In hierarchically composed components the labels are thus drawn from a richer set than just X. This requires a slight change in the definitions of separated and driven. In both cases P(X) should be replaced by $P(X_{tot})$.

The following properties show that the restrictions imposed on compo-

nents can be checked locally, i.e. for each local graph separately. For a more comprehensive treatment of restoring logic the reader is referred to REM 82, which includes the proofs of the properties.

PROPERTY 3.1. A component of which the local graph and all subcomponents are nonfighting is nonfighting.

PROPERTY 3.2. A component of which the local graph and all subcomponents are well-behaved is not necessarily well-behaved.

PROPERTY 3.3. A component is restoring if all its subcomponents are restoring and every vertex $j \in Y$ is driven in the local graph by $X_{int} \cup \{0,1\}$.

The following is an example of a restoring hierarchically composed component. Like in partially ordered computations, s.p signifies port p of subcomponent s.

```
com C(a?,b?,q!,qbar!):
    sub i0,i1: inverter
    a ∧ b → i0.in = 0
    a' ∨ b' → i0.in = i1.out
    a' ∧ b' → i1.in = 0
    a ∨ b → i1.in = i0.out
    q = i0.out
    qbar = i1.out
moc
```

The local graph of component C consists of eight labeled edges and two unlabeled ones. An unlabeled edge between x and y is specified as "x = y".

Consider the four lines representing labeled edges. They may be viewed as equations in 4 unknowns together with conditions (in a and b) under which they are valid. For given a and b there remain two equations. We, furthermore, have that the inputs and outputs of i0 and i1 are each other's inverse. Thus we have, for given a and b, a system of 4 equations in 4 unknowns. Together with the last two lines this gives 6 equations in 6 unknowns. If $a \neq b$ this system has two solutions. If a system has multiple

solutions there must have been an earlier moment at which it had one solution and that solution must be one of the multiple solutions. That unique solution is then chosen among the multiple ones. Thus history dependence (or storage or state) may be represented in this framework.

We interpret output q as the value stored. The output qbar is its complement. Component C is a Muller C-element with two inputs (SEITZ 80). The value stored can change only when the input values change. If the inputs are made equal the value is made equal to the inputs. Otherwise the value remains unchanged. Fig. 3 contains a diagram of the component. The C-element plays an important role in delay-insensitive circuits.

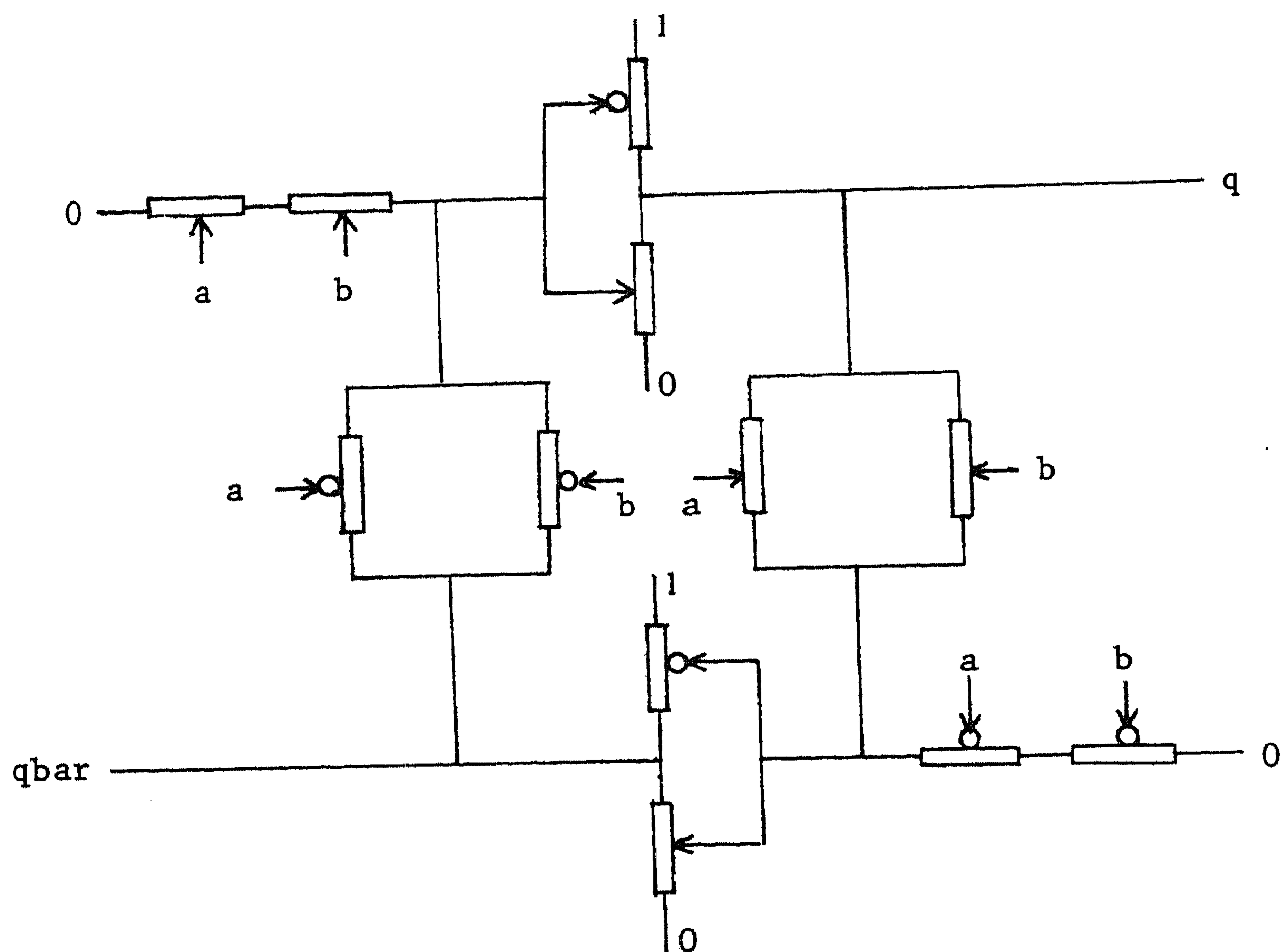


Fig. 3 Diagram of a C-element

As a last example of a restoring component we discuss a tally-circuit,

$\underline{com}$ $tally_0(out_0!)$; $out_0 = 1$ $\underline{moc}$

for i > 0:

<u>com</u> $tally_i(in_0?, \ldots, in_{i-1}?, out_0!, \ldots, out_i!)$.

  <u>sub</u> t: $tally_{i-1}$

  in_j = $t.in_j$ <u>forall</u> j <u>in</u> 0..i-2

  $in_{i-1} \rightarrow out_0 = 0$, $out_j = t.out_{j-1}$ <u>forall</u> j <u>in</u> 1..i

  $in_{i-1}' \rightarrow out_j = t.out_j$ <u>forall</u> j <u>in</u> 0..i-1, $out_i = 0$

<u>moc</u>

Component $tally_i$ has i input ports and i + 1 output ports. The line starting with in_j specifies i - 1 edges, one for every j satisfying $0 \leq j \leq i - 2$. The next line shows that we allow more than one edge to the right of an arrow. It specifies i + 1 connections, each of them labeled in_{i-1}. Component $tally_i$ counts the number of inputs that have value 1. If j ($0 \leq j \leq i$) inputs are 1 output out_j will be 1 and the other outputs 0. Figure 4 shows a diagram of $tally_3$. This is basically the same circuit as the tally in MEAD & CONWAY 80.

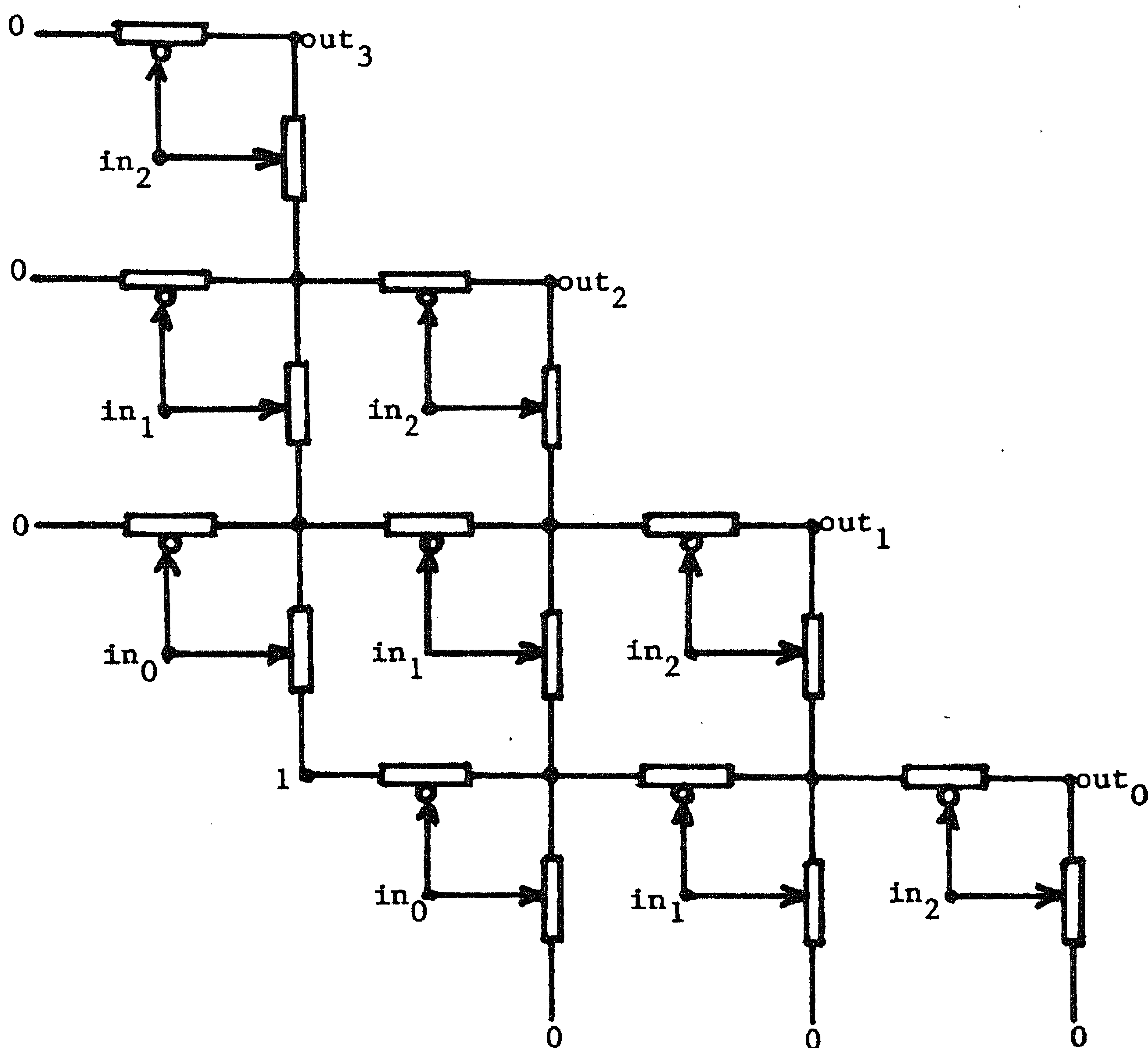


Fig. 4 Diagram of $tally_3$-circuit

4. DELAY-INSENSITIVE SIGNALING

In Section 2 we have become acquainted with the kind of computations we want to consider. Section 3 has given us a clear view of the kind of logic components we can build. But there is still an important gap left between them. This gap has mainly to do with what is customarily called "timing". If a component computes a result, and if that result is again input for another component, how then can we guarantee that the outputs are not used by the second component before they have assumed their values? The traditional method is to provide the components with clock signals that are generated at regular time intervals and that signal the completion of the preceding step. Since this requires estimating the connection delays, this approach is unsuitable for VLSI. In this section we discuss a different technique known as delay-insensitive signaling or self-timed signaling (SEITZ 80).

4.1. A composition operator expressing delay

We introduce a third composition operator: r-composition. It is similar to q-composition, but it expresses unbounded delay. There is a direction in delay: the sending of a signal will always precede its reception. To express this we introduce directed trace structures. (When appropriate, we refer to the trace structures defined in Section 2 as undirected trace structures.)

A directed trace structure is a triple $S = \langle T,A_0,A_1 \rangle$. A_0 and A_1 are disjoint finite subsets of Σ. $T \subset (A_0 \cup A_1)^*$. A_0 is called the *output alphabet* of S, and A_1 its *input alphabet*. The elements of A_0 and A_1 are called its *output symbols* and *input symbols*, respectively. When we say the alphabet of S, we mean $A_0 \cup A_1$.

We compose directed trace structures $S = \langle T,A_0,A_1 \rangle$ and $S' = \langle U,B_0,B_1 \rangle$ only when their alphabets satisfy $A_0 \cap B_0 = \emptyset$ and $A_1 \cap B_1 = \emptyset$. The p- and q-compositions of S and S' are p- and q-compositions of $\langle T,A_0 \cup A_1 \rangle$ and $\langle U,B_0 \cup B_1 \rangle$.

Any symbol a occurring in the intersection of the alphabets of S and S' is an output symbol in one and an input symbol in the other. To distinguish between a as an input symbol and a as an output symbol we introduce postfixed symbols. A symbol a can be postfixed with "!" or "?", yielding a! and a?, respectively. These are two distinct symbols.

Let A_0 and A_1 be disjoint sets of symbols. If B is an alphabet then $B/(A_0,A_1)$ is the alphabet obtained from B by replacing each symbol a in A_0 by the postfixed symbol a! and each symbol a in A_1 by a?. Trace set $T/(A_0,A_1)$ is similarly obtained from trace set T. Let S = <T,A> be an undirected trace structure. The undirected trace structure $S/(A_0,A_1)$ is then given by

$$S/(A_0,A_1) = \langle T/(A_0,A_1)\ ,\ A/(A_0,A_1)\rangle\ .$$

We introduce a special undirected trace structure DEL. Let a and b be two distinct symbols. DEL(a,b) is the trace structure with {a,b} as its alphabet and

$$\{t \in \{a,b\}^* \mid \#_a t' \geq \#_b t' \text{ for every prefix } t' \text{ of } t\}$$

as its trace set.

We now come to the definition of r-composition. Let $S = \langle T,A_0,A_1\rangle$ and $S' = \langle U,B_0,B_1\rangle$ be two directed trace structures. Consider the following undirected trace structure.

$$(4.1)\qquad \begin{array}{l} \langle T,A_0 \cup A_1\rangle/(A_0 \cap B_1, A_1 \cap B_0)\ \underline{q} \\ \langle U,B_0 \cup B_1\rangle/(B_0 \cap A_1, B_1 \cap A_0)\ \underline{q} \\ DEL(a_0!,a_0?)\ \underline{q}\ \ldots\ \underline{q}\ DEL(a_{n-1}!,a_{n-1}?) \end{array}$$

in which $\{a_0, \ldots ,a_{n-1}\} = (A_0 \cap B_1) \cup (A_1 \cap B_0)$, i.e. the intersection of the alphabets of S and S'.

The *r-composition* of S and S', denoted S $\underline{r}$ s', is trace structure (4.1) with its alphabet partitioned into the output alphabet $(A_0 \backslash B_1) \cup (B_0 \backslash A_1)$ and the input alphabet $(A_1 \backslash B_0) \cup (B_1 \backslash A_0)$.

The trace structures $DEL(a_i!,a_i?)$ express the delay between the sending of a_i and the reception of a_i.

<u>EXAMPLE 4.1.</u> Consider the program

(4.2) (z!; a; z!; b), (p; z?; q; z?)

(we have postfixed the symbols in the intersection of the alphabets to show

their types.) With the comma denoting q-composition (4.2) is equivalent to

(4.3) p; a,q;b

With r-composition (4.2) is equivalent to

(4.4) (a;b), (p;q)

Program (4.3) has two traces and (4.4) six, including the two of (4.2).

There are two problems associated with the delay as expressed by r-composition. The first one is that the trace set of DEL is not regular. DEL is the unbounded SYNC or, more precisely

$$DEL(a,b) = \bigcup_{k=1}^{\infty} SYNC_k(a,b)$$

Trace structure DEL(a,b) has as its states $\{[a^i] \mid i \geq 0\}$, which is an infinite set. By introducing unbounded delay we have, consequently, left the realm of the finite-state machines and, hence, that of the regular sets.

The second problem is that one might question the validity of the assumption that a connection has an unbounded buffering capacity. A wire, obviously, does not have this property. On the other hand, wires do exhibit delay. So we cannot simply dispense with r-composition.

Fortunately, matters are not as dim as they may look. We want to restrict our components in such a way that their r-composition equals their q-composition, thus restricting ourselves again to regular sets. This is reflected in the following definition. A composition of a collection of trace structures is called *delay-insensitive* if the collection's q-composition and r-composition yield the same trace sets. For delay-insensitive compositions the trace structures DEL(a_i!,a?) in (4.1) may, without affecting the composite, be replaced by $SYNC_1$(a_i!,a_i?). Hence, the connections need only accomodate buffering capacity 1, which seems a very reasonable assumption for wires.

It is possible to transform trace structures in such a way that their composition is delay-insentive without affecting their composite. We have actually already demonstrated this technique when we constructed component quinsem in Section 2.2. Compare this component with quinsem'. In the latter one the alphabets of the subcomponents are not disjoint: b0.p and b1.v are the same symbol. In quinsem this is implemented by adding the "acknowledge signal" q. The following property shows that this is a general technique.

Let $S = \langle T, A_0, A_1 \rangle$ be a directed trace structure and a a symbol with $a \in A_0 \cup A_1$. Then

$$S[a := a_0 a_1]$$

denotes the trace structure obtained from S by

(i) replacing in every trace of T the symbol a by the sequence $a_0 a_1$;
(ii) replacing a by a_0 in the alphabet that contains a;
(iii) adding a_1 to the alphabet that does not contain a.

(It is assumed that a_0 and a_1 are fresh symbols.)

$$S[\forall a \in C: a := a_0 a_1]$$

denotes the trace structure obtained from S by applying the transformations above for all symbols $a \in C$.

<u>PROPERTY 4.1.</u> Let $S = \langle T, A_0, A_1 \rangle$ and $S' = \langle U, B_0, B_1 \rangle$ be two directed trace structures. $A = A_0 \cup A_1$ and $B = B_0 \cup B_1$. Then

$$S \ \underline{q}\ S' =$$

$$S[\forall a \in A \cap B: a := a_0 a_1] \ \underline{q}\ S'[\forall a \in A \cap B: a := a_0 a_1] =$$

$$A[\forall a \in A \cap B: a := a_0 a_1] \ \underline{r}\ S'[\forall a \in A \cap B: a := a_0 a_1].$$

Thus we have found a way of making our compositions delay-insensitive. Applying this to Example 4.1 would give rise to the program

$$(z_0!;\ z_1?;\ a;\ z_0!;\ z_1?;\ b),\ (p;\ z_0?;\ z_1!;\ q;\ z_0?;\ z_1!)$$

which, both under q- and under r-composition, is equivalent to (4.3).

<u>4.2. Two delay-insensitive circuits</u>

In Section 3 we discussed logic components composed of switches and connections. The connections carry the values 0 and 1. We looked at how the output values depend on the input values, but we hardly discussed dynamic behaviour, i.e. transitions on the connections. A transition is a change in value. Let a be a point on a connection. A change from 0 to 1 in point a

(called a high-going transition) is denoted by $a\uparrow$, and a change from 1 to 0 (a low-going transition) by $a\downarrow$. Consider the program

(4.5) $\quad (v;\ t!)^*,\ (t?;\ w)^*.$

Under q-composition (4.5) yields $v;\ (w,v)^*$, i.e. $SYNC_2(v,w)$, under r-composition, however, (4.5) yields DEL(v,w). Property 4.1 tells us how to resolve this difference: we change (4.5) into

(4.6) $\quad (v;\ t!;\ t_1?)^*,\ (t_0?;\ t_1!;\ w)^*.$

But let there now be two connections, one for t_0 and one for t_1, and let the symbols t_i stand for transitions on these connections. Assume the connections to be low (carrying the value 0) initially. Since high-going and low-going transitions on the same connection alternate, (4.6) becomes

$$(v;\ t_0\uparrow!;\ t_1\uparrow?;\ v;\ t_0\downarrow;\ t_1\downarrow?)^*,\ (t_0\uparrow?;\ t_1\uparrow!;\ w;\ t_0\downarrow?;\ t_1\downarrow!;\ w)^*$$

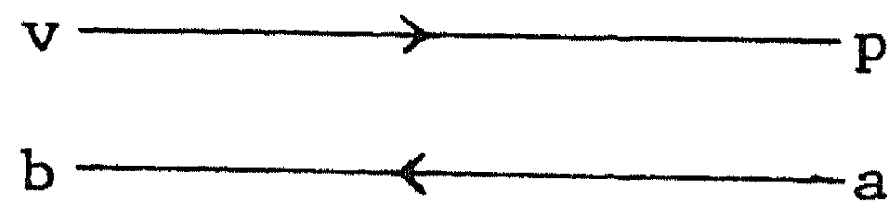
$t_i\uparrow!$ may be read as "drive connection t_i high" and $t_1\uparrow?$ as "observe connection t_i going high". This type of signaling is known as 2-cycle signaling: there are two transitions between successive v's or w's. A more customary way of delay-insensitive signaling is 4-cycle signaling;

$$(v;\ t_0\uparrow!;\ t_1\uparrow?;\ t_0\downarrow!;\ t_1\downarrow?)^*,\ (t_0\uparrow?;\ t_1\uparrow!;\ t_0\downarrow?;\ t_1\downarrow!;\ w)^*.$$

We have now four transitions between successive v's or w's. The 4-cycle scheme is, as we saw, not necessary to achieve delay-insensitivity, but it tends to make the circuits thus communicating simpler than with 2-cycle signaling. The reason for this is that every v (and w) takes place with the pair (t_0,t_1) of connections having the same value (cf. SEITZ 80).

In this section we design two components: a quick return linkage and a binary semaphore. As computations they are not very interesting, but they provide a good insight in the application of trace theory to delay-insensitive signaling. We shall employ 4-cycle signaling. As a consequence, we may use q-composition as our composition operator. We henceforth omit the question marks and exclamation points.

Consider again two connections with 4-cycle signaling. We give names to their endpoints



The 4-cycle signaling is expressed by the order

(4.7) $(v\uparrow;\ p\uparrow;\ a\uparrow;\ b\uparrow;\ v\downarrow;\ p\downarrow;\ a\downarrow;\ b\downarrow)^*$.

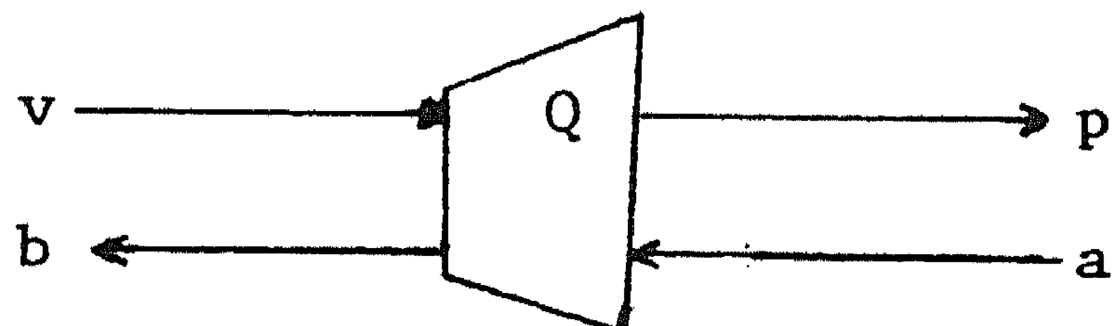
Expression (4.7) may be rewritten as

(4.8) $v\uparrow;\ (p\uparrow;\ a\uparrow;\ b\uparrow;\ v\downarrow;\ p\downarrow;\ a\downarrow;\ b\downarrow;\ v\uparrow)^*$.

We split in (4.8) the transitions at the left end of the connections from those at the right end, which yields - adding symbols to make it equivalent to (4.8)

(4.9)
$$v\uparrow;\ (t0;\ p\uparrow;\ a\uparrow;\ t1;\ t2;\ p\downarrow;\ a\downarrow;\ t3)^*,$$
$$(t0;\ t1;\ b\uparrow;\ v\downarrow;\ t2;\ t3;\ b\downarrow;\ v\uparrow)^*.$$

We can weaken the order expressed in (4.9) by inserting in the connections a so-called quick return linkage



(By drawing arrowheads in the connections we express whether a transition $a\uparrow$ stands for $a\uparrow!$ or for $a\uparrow?$.) The effect of the insertion of the quick return linkage is that $p\downarrow;\ a\downarrow$ does not have to precede $b\downarrow;\ v\uparrow$, i.e. that in expression (4.9) t3 is deleted:

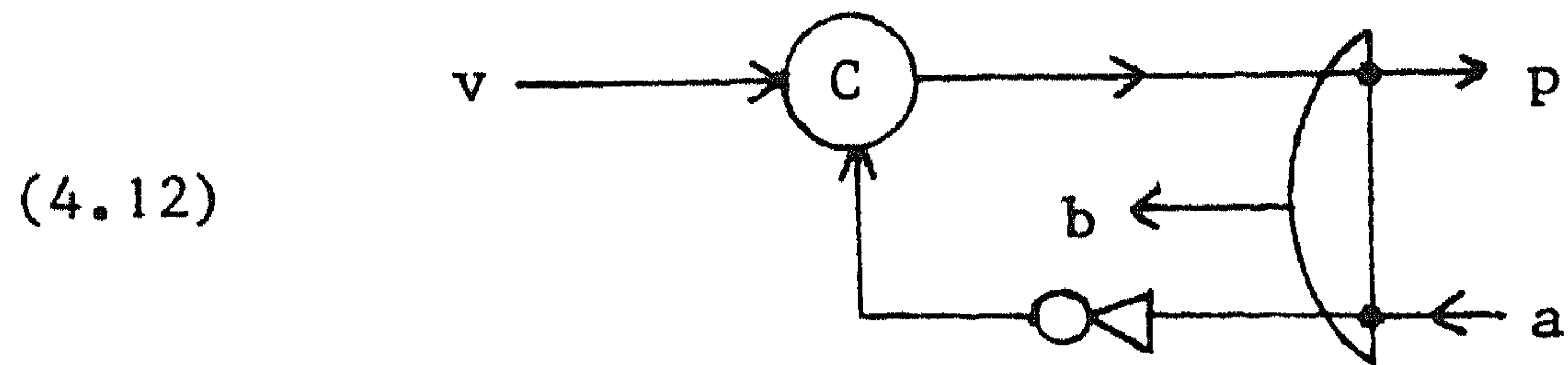
(4.10)
$$v\uparrow;\ (t0;\ p\uparrow;\ a\uparrow;\ t1;\ t2;\ p\downarrow;\ a\downarrow)^*,$$
$$(t0;\ t1;\ b\uparrow;\ v\downarrow;\ t2;\ b\downarrow;\ v\uparrow)^*.$$

Expression (4.10) is equivalent to

(4.11) $v\uparrow;\ (p\uparrow;\ a\uparrow;\ b\uparrow;\ v\downarrow;\ (p\downarrow;\ a\downarrow),\ (b\downarrow;\ v\uparrow))^*$.

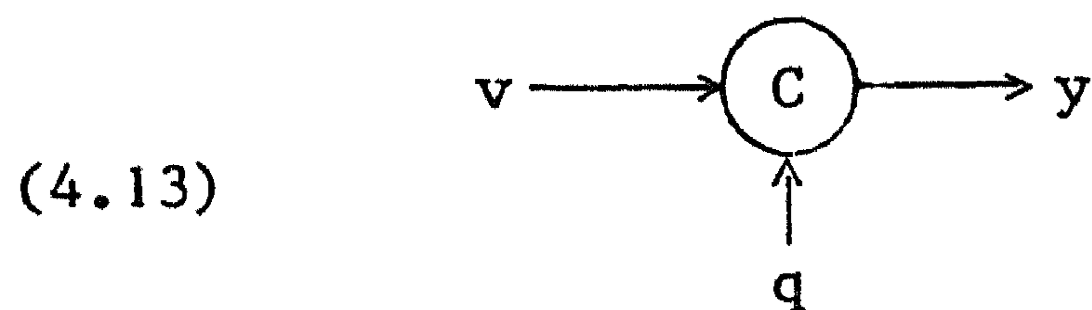
We show that Q can be implemented by

(4.12)



A circuit

(4.13)



denotes a C-element. It implements, as we saw in Section 3.2, the order

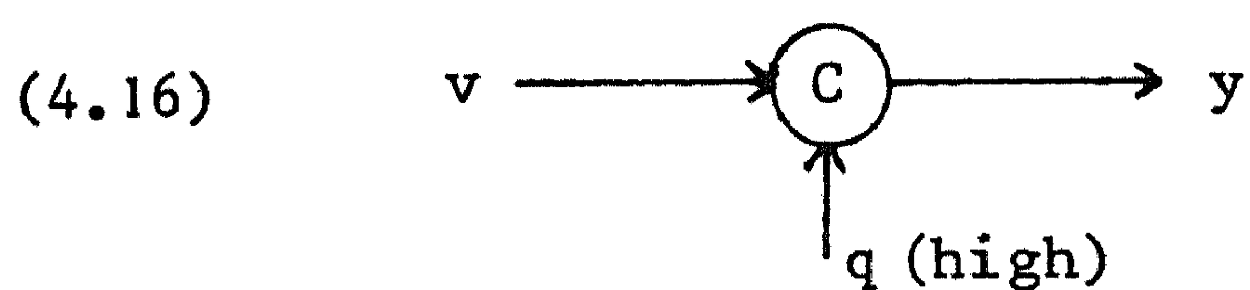
(4.14) $(v\uparrow,\ q\uparrow;\ y\uparrow;\ v\downarrow,q\downarrow;\ y\downarrow)^*$.

Expression (4.14) may be rewritten as

(4.15) $v\uparrow,q\uparrow;\ (y\uparrow;\ v\downarrow,q\downarrow;\ y\downarrow;\ v\uparrow,q\uparrow)^*$.

Consider the case that q is initially high:

(4.16)

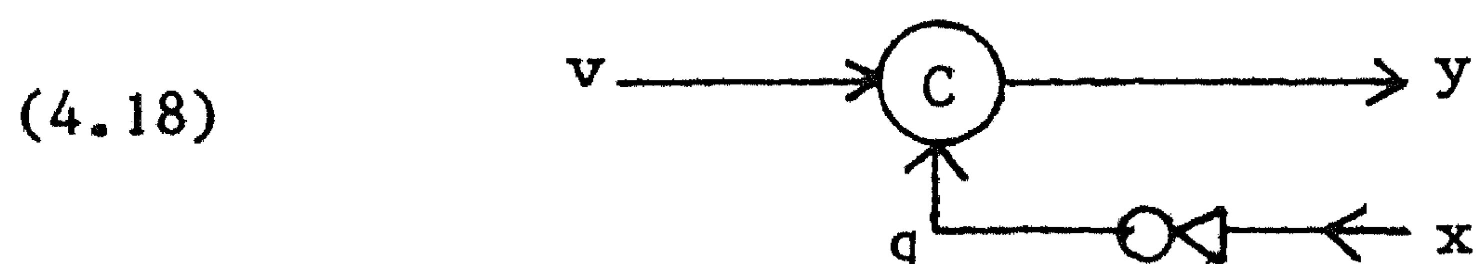


which corresponds to dropping the initial $q\uparrow$:

(4.17) $v\uparrow;\ (y\uparrow;\ v\downarrow,q\downarrow;\ y\downarrow;\ v\uparrow,q\uparrow)^*$.

Next, let circuit (4.16) be extended with an inverter at q:

(4.18)

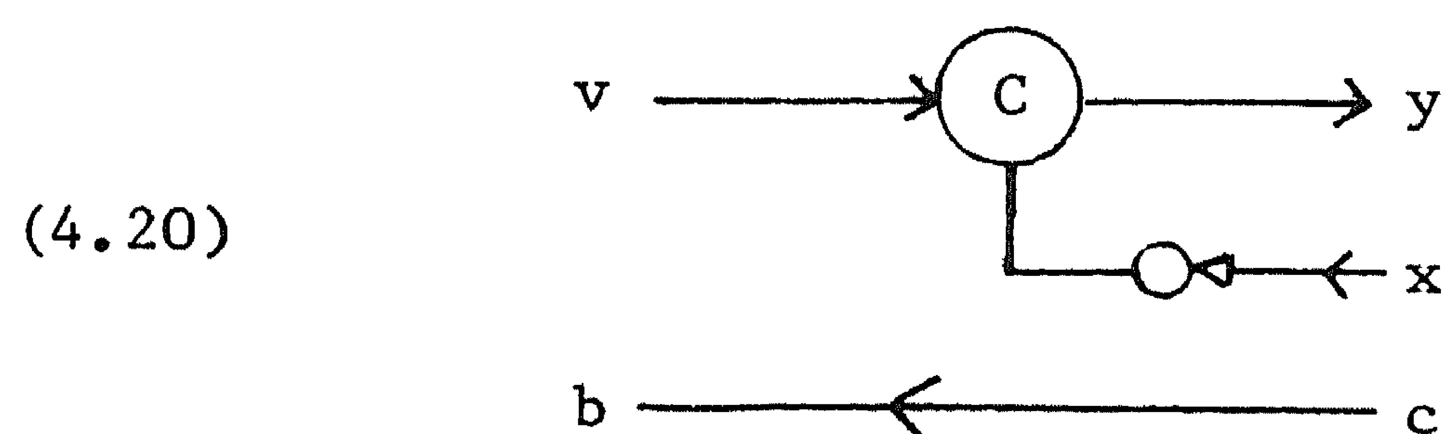


The expression for circuit (4.18) is obtained from (4.17) by replacing $q\downarrow$ and $q\uparrow$ by $x\uparrow$ and $x\downarrow$, respectively:

(4.19) $v\uparrow;\ (y\uparrow;\ v\downarrow,x\uparrow;\ y\downarrow;\ v\uparrow,x\downarrow)^*$.

We extend circuit (4.18) with a wire

(4.20)



The order for the wire is

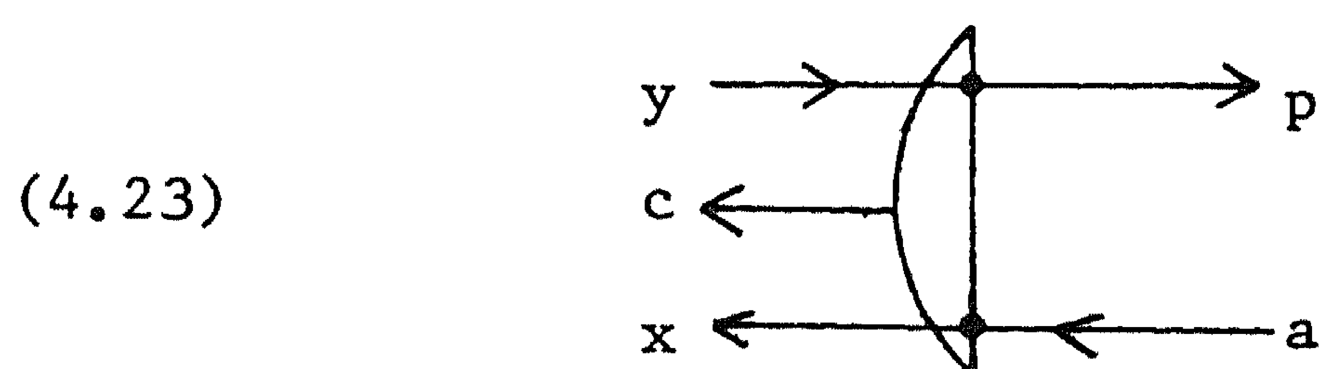
(4.21) $(c\uparrow;\ b\uparrow;\ c\downarrow;\ b\downarrow)^*$.

Circuit (4.20) has a left and a right environment. With both environments 4-cycle signaling is employed. This yields the following combination of (4.19) and (4.21) as the order for circuit (4.20):

(4.22) $v\uparrow;\ (y\uparrow;\ (c\uparrow;\ b\uparrow;\ v\downarrow),\ x\uparrow;\ y\downarrow;\ (c\downarrow;\ b\downarrow;\ v\uparrow),x\uparrow)^*$.

A circuit

(4.23)

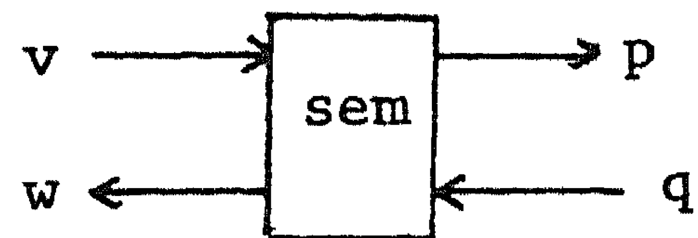


denotes an and-element with input reproduction. It has a left and a right environment, using 4-cycle signaling with both environments: the order for the left environment is $(y\uparrow;\ c\uparrow,x\uparrow;\ y\downarrow;\ c\downarrow,x\downarrow)^*$ and for the right environment $(p\uparrow;\ a\uparrow;\ p\downarrow;\ a\downarrow)^*$. The order for circuit (4.20) is the following combination of these two orders

(4.24) $(y\uparrow;\ p\uparrow;\ a\uparrow;\ c\uparrow,x\uparrow;\ y\downarrow;\ c\downarrow,(p\downarrow;\ a\downarrow;\ x\downarrow))^*$.

Circuit (4.12) is the composition of circuits (4.20) and (4.23). The q-composition of their expressions, (4.22) and (4.24), is (4.11), which shows that circuit (4.12) implements (4.11).

We next discuss the design of a binary semaphore. A binary semaphore is a component with a left and a right environment.



It implements the order expressed by

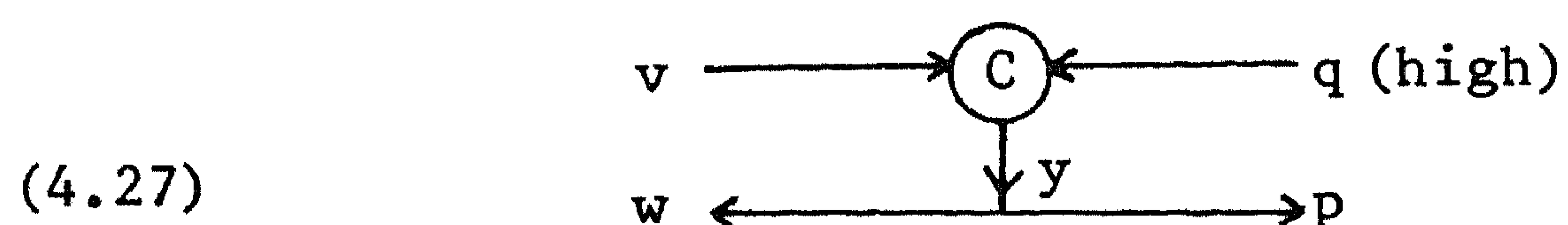
(4.25) $(v\uparrow;\ t0;\ w\uparrow;\ v\downarrow;\ w\downarrow)^*,\ (t0;\ p\uparrow;\ q\uparrow;\ p\downarrow;\ q\downarrow)^*.$

Expression (4.25) may be rewritten as

(4.26) $v\uparrow;\ (t0;\ w\uparrow;\ v\downarrow;\ w\downarrow;\ v\uparrow)^*,\ (t0;\ p\uparrow;\ q\uparrow;\ p\downarrow;\ q\downarrow)^*.$

Consider again circuit (4.16). We split output y into outputs w and p:

(4.27)



The order for circuit (4.27) is obtained by replacing in (4.17) $y\uparrow$ and $y\downarrow$ by $w\uparrow,p\uparrow$ and $w\downarrow,p\downarrow$, respectively:

(4.28) $v\uparrow;\ (w\uparrow,p\uparrow;\ v\downarrow,q\downarrow;\ w\downarrow,p\downarrow;\ v\uparrow,q\uparrow)^*.$

Next we split the environment of circuit (4.27) into a left and a right environment, each of which uses 4-cycle signaling: the order for the left environment is $v\uparrow;\ (w\uparrow;\ v\downarrow;\ w\downarrow;\ v\uparrow)^*$ and for the right environment $(p\uparrow;\ q\downarrow;\ p\downarrow;\ q\uparrow)^*$. The result of this splitting is that the environment as a whole imposes less order. The following orders, that were present in (4.28), have been lost:

$w\uparrow;\ q\downarrow$

$p\uparrow;\ v\downarrow$

$w\downarrow;\ q\uparrow$

$p\downarrow;\ v\uparrow$.

Deleting these orders from (4.28) yields

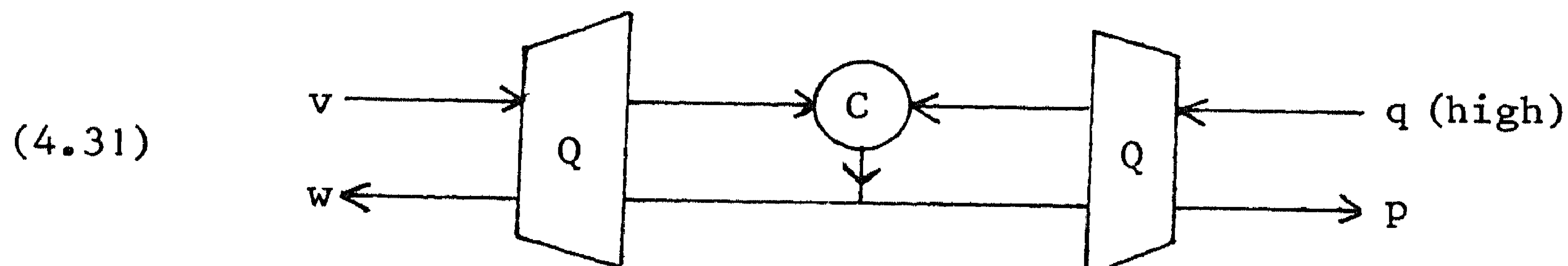
(4.29) $v\uparrow; ((w\uparrow;\ v\downarrow), (p\uparrow;\ q\downarrow);\ (w\downarrow;\ v\uparrow), (p\downarrow;\ q\uparrow))^*.$

Expression (4.29) is equivalent to

(4.30)
$$v\uparrow;\ (t0;\ w\uparrow;\ v\downarrow;\ t1;\ w\downarrow;\ v\uparrow)^*.$$
$$(t0;\ p\uparrow;\ q\downarrow;\ t1;\ p\downarrow;\ q\uparrow)^*.$$

Expression (4.30) exhibits more order than (4.26). But we know what we can do about that: add a quick return linkage. The effect of adding a quick return linkage on v and w is that $p\uparrow;\ q\downarrow$ does not have to precede $w\downarrow;\ v\uparrow$, i.e. that in expression (4.30) $t1;\ w\downarrow;\ v\uparrow$ is replaced by $t1, (w\downarrow;\ v\uparrow)$. Another quick return linkage on p and q replaces $t1;\ p\downarrow;\ q\uparrow$ by $t1, (p\downarrow;\ q\uparrow)$. Thus the circuit

(4.31)



implements

(4.32)
$$v\uparrow;\ (t0;\ w\uparrow;\ v\downarrow;\ t1, (w\downarrow;\ v\uparrow))^*,$$
$$(t0;\ p\uparrow;\ q\downarrow;\ t1, (p\downarrow;\ q\uparrow))^*.$$

Expression (4.32) is equivalent to (4.26), which shows that circuit (4.31) is a binary semaphore.

Like we showed with component trisem in Section 2.2, we can connect two of these binary semaphores to obtain a ternary semaphore, or k binary semaphores to obtain a (k+1)-ary semaphore.

5. CONCLUSIONS

In the preceding sections we have discussed a number of topics associated with the translation of components denoting partially ordered computations into delay-insensitive circuits. There are also a number of topics we have not touched upon or only touched upon lightly. We outline some of the latter topics in this section.

There are a number of ways in which the translation of components into circuits can be achieved. We discuss two of them.

Some circuits can be translated into regular structures known as programmable logic arrays (PLA's). As an example we consider the full-adder element discussed in Section 2.3 (Example 2.8). We can partition its alphabet into an input and an output alphabet. Such a partition may be given to start with, but there are also efforts being made to derive the partition from the trace set. We give an example of such a derivation that seems applicable in simple cases.

Let $S = \langle T,A \rangle$ be a trace structure. We call two symbols a and b in A related if

$$\{u \in A^* \mid ua \in \mathrm{PREF}(T)\} = \{u \in A^* \mid ub \in \mathrm{PREF}(T)\}.$$

This is an equivalence relation on A. We call each equivalence class containing at least two symbols an input and the symbols in such a class input symbols. The singleton equivalence classes are then the outputs and the symbols in them output symbols.

The definition above yields for the full-adder element three binary inputs: (a0,a1), (b0,b1), and (c0,c1). The other symbols are output symbols. For each output symbol we list the input symbols that precede it, i.e. those that are separated from it by at least one semicolon:

$$d0:\ a0 \wedge b0 \vee a0 \wedge b1 \wedge c0 \vee a1 \wedge b0 \wedge c0$$

$$d1:\ a1 \wedge b1 \vee a0 \wedge b1 \wedge c1 \vee a1 \wedge b0 \wedge c1$$

$$s0:\ a0 \wedge b0 \wedge c0 \vee a1 \wedge b1 \wedge c0 \vee a0 \wedge b1 \wedge c1 \vee a1 \wedge b0 \wedge c1$$

$$s1:\ a0 \wedge b0 \wedge c1 \vee a1 \wedge b1 \wedge c1 \vee a0 \wedge b1 \wedge c0 \vee a1 \wedge b0 \wedge c0$$

Seitz shows an almost delay-insensitive PLA for a full-adder element on p. 251 of SEITZ 80. The 14 terms in the four lines above correspond exactly to the 14 crossings in his PLA at which the horizontal and vertical wires are connected. Seitz's realization may thus be derived directly from our program text.

For more complicated components - components with nested repetition, for example - we can draw upon work done on recognizing regular languages (FLOYD & ULLMAN 80). There is a difference between recognizing a regular

language and executing a component: when executing a component, i.e. selecting a trace, the input symbols are the only ones to be recognized. At the appropriate positions in the trace recognized the output symbols in that trace must be generated. These output symbols are then again inputs to and recognized by other components.

An interesting method of recognizing regular languages is described in FOSTER & KUNG 81. The machinery they propose consists of so-called building blocks that communicate in a pipe-lined fashion. Since such a scheme does not require global communication, it seems well-suited for a delay-insensitive implementation. A necessary extension of their machinery in order to use it for the execution of our components is the introduction of a building block that corresponds to the comma.

We have chosen trace theory as the basis of our computations, and we have showed how states (and state transitions) can be derived from trace sets. A more traditional approach is to start with states. Such an approach seems well-suited for clocked systems: the clock separates the computation into a number of steps, each step leading from one state to the next. In delay-insensitive circuits, however, there do not have to be moments at which the circuit is in a well-defined state. We are dealing with events that are only partially ordered. Trace theory seems to be well-suited as a basis for such computations. In BROCK & ACKERMAN 81 it is shown that defining the semantics of (nondeterministic) components as functions from input streams to sets of output streams is not a good approach, since it does not reflect the order between individual items in different streams. They exhibit this shortcoming in a number of approaches described in the literature. Trace theory does not have this defect.

There are at least three extensions to the material presented that need further attention. First of all, we need more theorems on trace structures. These theorems must enhance our capability of designing partially ordered computations and arguing about them. They must be aimed at leaving the realm of the individual traces. A nice example of such a theorem is Property 2.11. It defines the net effect of the composite in terms of the net effects of the parts. Finding such theorems requires good ways of characterizing net effects of computations. The characterization of stacks in terms of well-nested sequences (Example 2.7) was an effort in that direction.

Secondly, we need to study the problem of laying out transistor-con-

nection diagrams (so-called schematics) in the plane. This mapping must be done by a compiler, without interference or consultation of the designer. The fact that the mapping problem is still ill-understood is the main driving force behind the current popularity of graphical "design tools". Such design tools are usually design obstacles in the sense that they prevent the designer from ignoring the physical realization.

Thirdly, we need to extend the notation in which we specify our components. A good extension would be the introduction of values and ports via which these values are communicated. It would make our notation look more like the one discussed in HOARE 78a. Values and ports have types. The declaration of a type defines a method of representing values and ports of that type in terms of symbols. Such a type could, for example, be 16-bit integer. The declaration specifies, among other things, whether the bits of such an integer are communicated serially of in parallel. The components we introduce will thus give us new modes of expression. By a proper choice of components we want to arrive at a mode of expression that one would customarily call a "higher level programming language".

6. ACKNOWLEDGEMENTS

It would have been impossible to write these notes without the cooperation of Jan van de Snepscheut. Many of the concepts introduced are either his or took shape during and as a result of our scientific collaboration.

Carver Mead introduced me to the fascinating world of VLSI. In many long discussions we tried to grasp the essence of computing machinery. His insights have greatly contributed to my understanding of VLSI.

Chuck Seitz taught me the art of dealing with delays. Virtually all I known of delay-insensitive signaling can in some way or another be traced back to him. Charles Molnar and Jo Ebergen are gratefully acknowledged for deepening my insight in delay-insensitive computations.

Arjen Lenstra and Paul Vitányi suggested part of the mathematical formalism used in these notes. Jan Tijmen Udding pointed out a number of shortcommings in Section 4.2. While preparing these notes I discussed their contents with Alain Martin, Randy Bryant, and Young-il Choo. The material has also served as the subject of attention at Edsger W. Dijkstra's Tuesday Afternoon Club. Both occasions resulted in a number of improvements. Acknowledgements are also due to the members of the MC-THE VLSI Working Group,

who critically evaluated most of the material presented.

Eindhoven University of Technology and California Institute of Technology are gratefully acknowledged for giving me the opportunity to commute regularly between these two institutes. The combination of their scientific atmospheres proved an excellent breeding-place for the research reported.

7. REFERENCES

BROCK, J.D. & ACKERMAN, W.B. (1981), *Scenarios: a model of non-determinate computation*, Computation Structures Group Memo 206, Massachusetts Institute of Technology, Cambridge, Mass. (Also in the proceedings of the International Colloquium on Formalization of Programming Concepts, Peniscola, Spain. Lecture Notes in Computer Science 107, Springer-Verlag, Berlin)

CAMPBELL, R.H. & HABERMANN, A.N. (1974), *The specification of process synchronization by path expressions*, in "Operating Systems" (Eds. E. Gelenbe & C. Kaiser), pp. 89-102. Lecture Notes in Computer Science 16. Springer-Verlag, Berlin.

CLARK, W.A. (1980), *From electron mobility to logical structure: a view of integrated circuits*, ACM Comp. Surveys 12,3, pp. 325-356.

FLOYD, R.W. & ULLMAN, J.D. (1980), *The compilation of regular expressions into integrated circuits*, in "Proceedings of the 21st Annual Symposium on Foundations of Computer Science", pp. 260-269. IEEE Computer Society. (Also in Journal of the ACM, 29,3, pp. 603-622)

FOSTER, M.J. & KUNG, H.T. (1980), *Recognize regular languages with programmable building-blocks*, in "VLSI 81 - Very Large Scale Integration", pp. 75-84, Academic Press, London.

GINSBURG, S. (1966), *The Mathematical Theory of Context-free languages*. McGraw-Hill, New York, N.Y.

HOARE, C.A.R. (1978), *Towards a theory of communicating sequential processes*, Preliminary Report, Oxford University, Oxford.

HOARE, C.A.R. (1978a), *Communicating sequential processes*, Comm. ACM. 21,8, pp. 666-677.

HOPCROFT, J.E. & ULLMAN, J.D. (1969), *Formal Languages and their Relation*

to Automata, Addison-Wesley, Reading Mass.

LAUER, P.E. (1981), *Synchronization of concurrent processes without globality assumptions*, ACM Sigplan Notices 16,9, pp. 66-80.

MARTIN, A.J. (1981), *An axiomatic definition of synchronization primitives*, Acta Informatica 16, pp. 219-235.

MEAD, C. & CONWAY, L. (1980), *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass.

MILNER, R. (1980), *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92, Springer-Verlag, Berlin.

REM, M. (1979), *Mathematical aspects of VLSI design*, In "Caltech Conference on VLSI" (Ed. C.L. Seitz), pp. 55-64, California Institute of Technology, Pasadena, Calif.

REM, M. (1982), *On the design of restoring logic circuitry*, Eindhoven University of Technology, Eindhoven, Netherlands. (To appear in the proceedings of the 1982 CREST Summerschool on VLSI, Bristol, U.K.)

SEITZ, C.L. (1979), *Self-timed VLSI systems*, In "Caltech Conference on VLSI" (Ed. C.L. Seitz), pp. 345-355, California Institute of Technology, Pasadena, Calif.

SEITZ, C.L. (1980), *System timing*, In MEAD & CONWAY 80, pp. 218-262.

STUCKI, M.J. & COX, JR. J.R. (1979), *Synchronization strategies*, In "Caltech Conference on VLSI" (Ed. C.L. Seitz), pp. 375-394, California institute of Technology, Pasadena, Calif.

VAN DE SNEPSCHEUT, J.L.A. (1982), *An inventory of trace theory*, Internal report JAN83a, Eindhoven University of Technology, Eindhoven, Netherlands.