Some reflections on the implementation of trace structures

Some reflections on the implementation of trace structures

by

Rob Hoogerwoord

86/03

October 1986

Some reflections on the implementation of trace structures

contents

0. Introduction

In chapter 6 of his dissertation [0] Anne Kaldewaij presents an implementation of a trace structure called $SEM_1(a,b)$ in the form of a digital circuit (See fig. 6.2 in [0]). At first sight I liked this circuit very much for its simplicity and its seeming symmetry. As far as I am concerned,the simplicity is completely beyond dispute but the symmetry is not there. This is a little surprising because, by its specification, $SEM_1(a,b)$ is symmetric in a and b in the sense that $SEM_1(a,b) = a;SEM_1(b,a)$. The asymmetry due to the fact that a is the event to occur first I consider a minor one in the sense that it should be possible to derive a symmetric circuit in which the value of the initial state would account for the asymmetry. At closer scrutiny I became even more dissatified with the circuit for various reasons.

Firstly, the implementation is such that a is "passive" and b is "active". This is a deliberate choice of the designer, but now it is not immediately clear whether and, if so, how a circuit with, say, a and b both passive can be derived from this circuit in a simple way, e.g. without the introduction of another C-element. Moreover, it might very well be that the asymmetry thus introduced is also responsible for the asymmetry mentioned above. Therefore, I decided to try to design a new circuit, preferably as simple as the original one, with a and b of the same "type" and retaining the symmetry. The activities arising from this decision are the immediate cause for writing this note.

Secondly, the introduction of the, so-called, internal forks struck me as some sort of measure ad hoc to repair a delay-sensitive circuit that is supposed to be delay-insensitive. Initially, I thought that it would be possible to avoid the use of such forks altogether, thus simplifying the reasoning about such circuits. This, however, proved to be too optimistic an attitude. In the designs presented in this note internal forks do occur too.

Thirdly, although the circuit is correct in the sense that it indeed implements $SEM_1(a,b)$ it is a little bit overspecific because the transitions on the various wires are more strictly ordered than necessary, again in an asymmetric way. This is only

a minor inconvenience but it might be a symptom that the way of derivation is not yet what it could be. For example: according to the rules of the game the following is a valid sequence of events but the circuit will never produce it:
ai↑;ao↑;ai↓;ao↓;bo↑;bi↑;ai↑;ao↑ . The circuit is such that the second ao↑ is always preceded by bi↓ .

The latter two reasons for dissatisfaction probably are due to the circumstance that we do not know yet what are the rules and proof obligations by means of which correct circuits are to be derived. In this respect I have not very much to add but nevertheless I was able to "derive" a circuit satisfying me better; moreover, its derivation required the invention of a generalisation of the well-known C-element which I, therefore, will call G-element. As the implementation of a G-element is equally simple as the implementation of a C-element, it might very well be that G-elements are more useful than C-elements. For such claims, however, it is still too early.

This note is a report of my experiments. I have no other pretensions whatsoever with it than to share with others my experiences because they excited me very much. In fact, they still do. The discussion will be rather informal and operational: firstly, I am only an amateur circuit designer for whom his experience is probably more of a nuisance than of great help, and, secondly, formalisation probably would lead to a dissertation all by itself. I shall try to be somewhat explicit about the rules I use, especially when they differ from the ones adopted in [0], but I will not always -- be able to -- explain why I do so.

Finally, I must confess that I have not read the paper by Alain J. Martin [1]. Hence, it may very well happen that some of the ideas developed here already occur in Alain's paper. Well, let it be.

1. Four-phase hand-shaking revisited

Each event x of the trace structure to be implemented is represented by two wires xi and xo , where xi is an input to and xo is an output from the circuit implementing the trace structure. The event x takes place when on the pair xi,xo the four-phase hand-shaking protocol takes place. This protocol is a sequence of 4 "smaller" events, called "transitions", in such a way that any two successive transitions occur on different wires. This definition leaves room for exactly two such sequences, namely: xi;xo;xi;xo and xo;xi;xo;xi , in which we used the names of the wires to denote transitions on those wires. Because xi is an input the happening of the event x is, to some extent, initiated by the environment when the first sequence is used. When the second one is used the circuit initiates the event, for xo is an output. For any event x always the same sequence is used; is it the first one than the event is called "passive", is it the second one than the event is called "active". All this is completely in accordance with [0] but we would like to stress that the distinction of active and passive events is an unavoidable consequence of the decision to use four-phase hand-shaking. Notice that, for this discussion, the directions -- up or down -- of the transitions are irrelevant: at any moment in time on each wire exactly one transition is possible, viz. from the current value to the other value.

As the wires connecting the circuit and its environment may exhibit positive delays the protocol actually comprises 8 events: 4 on the side of the circuit and 4 on the environment side. See also figure 1.0. As these events are separated in time we have to decide at which points of time we are allowed to say that x has happened; similarly, of course, we have to decide when we are allowed to say that x has not (yet) happened. Herewith, we adopt a convention that is slightly more restrictive than the one used in [0]. Surprisingly enough, the circuits proposed in [0] comply with this convention too, so it probably is not that bad.

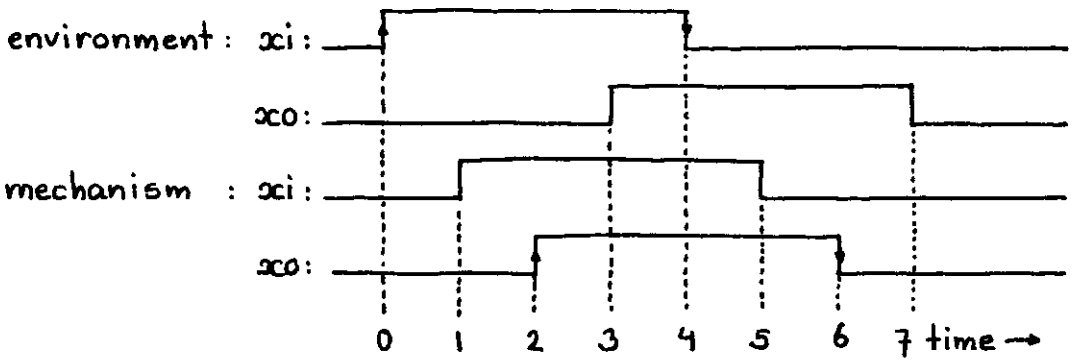

fig. 1.0: four-phase hand-shake for passive x

Firstly, an event may only happen if both the circuit and the environment are "prepared" to let it happen. The earliest possible moment at which we could say that, for passive x, both the circuit and the environment are prepared to let x happen is -- see figure 1.0 -- the transition labelled 2, i.e. transmission of xo↑ by the circuit.

Secondly, the transition labelled 5 , i.e. reception of xi↓ , is of interest: at this moment, the circuit "knows" that the environment has received the transition xo↑ ; hence, from moment 5 onwards we must accept that x has happened.

What about the interval from 2 to 5 ? Well, we decide that we do not care. From the point of view of the circuit we say: x has happened somewhere in between xo↑ and xi↓ . Similarly, for active events y: y has happened somewhere in between -- transmission of -- yo↑ and -- reception of -- yi↑ . If the trace structure is such that two events should be ordered than we play it safely and we require the circuits to be such that the corresponding intervals of time are disjoint and that they are appropriately ordered. Notice that this is sufficient because trace structures only define the relative orders of events: each event is now represented by an interval of time.

Summarising the above, the following rules of implementation may be formulated:

for passive x: "x precedes y" ≡ "xi↓ precedes yo↑" ,
for active x: "x precedes y" ≡ "xi↑ precedes yo↑" .

Notice that the -- in our opinion awkward -- notions "time" and "moment" have disappeared from these rules. This is a comforting observation.

2. Active or passive?

In the introduction we announced that we are heading for a circuit for $SEM_1(a,b)$ that is symmetric in a and b . For this purpose, it seems wise to let a and b be of the same "type", i.e. both passive or both active. In [0] the author shows -- see Theorem 6.3.0 and Theorem 6.3.1 -- that transforming a passive event into an active one is much easier than the other way round; the latter transformation requires an additional C-element whereas the former one can be realised by the addition -- or removal! -- of a single inverter. Therefore, we decide that a and b shall be passive.

(Aside: if each active event would be implemented as a passive event transformed using Theorem 6.3.0, then the transformation of an active event into a passive one also requires addition of a single inverter only. This looks like a considerable simplification but Warning 6.3.1 [0] might be an indication that this attitude is a little too naive.)

The trace structure of $SEM_1(a,b)$ is -- the prefix closure of -- $(a;b)*$. In the following "handshaking expansion" we have, in order to avoid overspecification, separated the hand-shaking protocols for a and b from each other and from the requirement that a and b must alternate:

$(ai\uparrow;ao\uparrow;ai\downarrow;ao\downarrow)*$, $(bi\uparrow;bo\uparrow;bi\downarrow;bo\downarrow)*$,
$(ai\downarrow;bo\uparrow)*$, $(ao\uparrow;(bi\downarrow;ao\uparrow)*)$.

The last "weavand", of course, can be rewritten as: $(ao\uparrow;bi\downarrow)*$. Notice that all essential information is in the last two weavands: if it were only the two hand-shaking protocols that mattered then some length of wire would suffice for the implementation, or, to put it differently, the last two weavands express the interaction of the events a and b .

The above trace structure may be taken as a specification of the circuit in terms of transitions. It now is a rather obvious step to transscribe this trace structure into the following program:

*[[ai];ao↑;[¬ai];ao↓] , *[[bi];bo↑;[¬bi];bo↓] ,
*[[¬ai];bo↑] , *[ao↑;[¬bi]] .

This program, however, is wrong: if, initially, all wires are false and, subsequently, the transition bi↑ occurs, then all guards preceding bo↑ are true; hence, this programs allows b to happen before a , which is in conflict with the specification. The problem is that we should distinguish transitions from values. The sequence ai↓;bo↑ prescribes that a down-going transition on ai should precede bo↑ , whereas [¬ai];bo↑ prescribes something like: "only if ai has value false bo↑ may occur", or, even weaker: "only if ai has been observed to have had value false bo↑ may occur". (The latter one, indeed, is very ugly and is usually avoided by some requirement of monotonicity). Apparently, ai↓;bo↑ and [¬ai];bo↑ are not the same. On the other hand, when taken in isolation, nothing is wrong with *[[ai];ao↑;[¬ai];ao↓] as a program for (ai↑;ao↑;ai↓;ao↓)* ; from that program the circuit consisting of a single wire is easily derived. So, a good question seems to be: under what conditions may transitions on an input wire be treated as if they were values? For the time being we shall not try to answer this question.

3. Intermezzo: a transition detector

According to the specification derived in the previous section ai↑;ao↑;ai↓;ao↓ is a valid trace of the circuit. After this trace all wires have the same values as they had initially, but the circuit is in a state different from the initial state: after this, b may happen. As these two states cannot be distinguished by means of the values of the wires, the introduction of additional variables is indicated. We shall call such variables "state variables".

The above observation is not at all surprising: the two states mentioned correspond to the two states of $SEM_1(a,b)$. What is surprising, however, is that two states -- and, hence, a single binary variable -- are insufficient for the implementation of $SEM_1(a,b)$. At least two state variables are needed and initially it was unclear why this would be so. Therefore, we carried out the following little exercise.

Suppose we wish to design a circuit with one input wire, a , and one output wire, x . Initially, both wires are false. For given constant k, $k \geq 0$, the output is to become true if on the input at least k transitions have occurred, after which x remains true forever. We think that such a circuit will have at least $k + 1$ states. Here, we shall treat the special case $k = 2$ only. We do so by applying the old technique of constructing a state transition table. We silently assume that the output is a function of the states; hence, we may confine our attention to the changes of the state and deal with the output later. The states are numbered using the naturals; the state itself is represented by the abstract state variable s ; the initial state is 0 :

a / s	0	1
0	0	1
1	2	1
2	2	2

Both the first and the second transition on a give rise to a change of state, hence the introduction of 3 states. The output x can now be expressed in the state as follows: $x \equiv (s=2)$. If one tries to

reduce the number of states by identifying two of them one will discover that this is impossible without violation of the specification: either x becomes true too early or it remains false forever. Apparently, the circuit must count the transitions that have occurred until the k-th transition. The moral of the story is that it seems impossible to exploit the circumstance that any two successive transitions on the same wire have opposite directions.

The abstract state variable s can be implemented by two binary state variables x and y by means of the following state assignment (such that ¬x automatically "is" the output):

$$s=0 \equiv \neg x \wedge \neg y ,$$
$$s=1 \equiv \neg x \wedge y ,$$
$$s=2 \equiv x \wedge y .$$

From this a circuit is easily constructed, but this is none of our concerns here (See, however, section 6).

What is the relevance of the above to our original problem? Well, if we project the trace ai↑;ao↑;ai↓;ao↓ on the inputs we obtain ai↑;ai↓ and we conclude that the state of the circuit after this trace can only be "reached" after the occurrence of 2 transitions on ai ; hence, we need at least 3 states, for the implementation of which at least 2 binary state variables are required.

(Historical aside: When I started this enterprise I was not able to derive a circuit for $SEM_1(a,b)$ in the way presented in the next sections: I did not know what rules to use and chapter 6 in [0] did not seem to provide much help either. Therefore, I did it the old-fashioned way. The result was the following state transition table; again, 0 is the initial state:

ai	0	1	1	0
bi	0	0	1	1
s				
0	0	1	1	0
1	2	1	1	2
2	2	2	3	3
3	0	0	3	3

Again, the outputs can be expressed in the state, as follows:
ao ≡ (s=1) , and: bo ≡ (s=3) . Notice the symmetry! The state can be represented by two state variables x and y . After having chosen the state assignment for the initial state and adopting the rule that on each transition at most one variable changes its value, the whole state assignment is fixed (but for permutation of x and y). The rest of the derivation is quite standard -- Karnaugh maps and all that -- and, therefore, is omitted here. It was during this activity that I discovered the G-elements to be discussed later.

Surprisingly enough the resulting circuit is completely identical to the circuit we shall derive in the next section.
(Hint: use x∧¬y ≡ (s=0) , x∧y ≡ (s=1) , and so on).
(End of historical aside).

4. Is this a derivation?

Now it is about time to do some useful work, i.e. to derive a circuit for $SEM_1(a,b)$. Before doing so, however, we shall state the rules of our game.

Firstly, we need some general properties of the "elements" used. An element has a single output and some -- zero or more -- inputs. An element with output x is completely specified by means of two boolean expressions, B0 and B1 say; the specification is, as in [0] and [1], denoted as follows:

$$B0 \rightarrow x\uparrow \ ,$$
$$B1 \rightarrow x\downarrow \ .$$

B0 and B1 are expressions in the names of the element's inputs; we shall call B0 and B1 the "guards" of the element. In order to avoid conflicts on what the value of the output should be we require each element to satisfy the following "rule of disjointness":

$$\neg B0 \vee \neg B1 \ , \text{ for all values of the inputs.}$$

It is our impression that this rule has heuristic value too.

A particularly effective and simple way to enforce the rule of disjointness is the following one: strengthen the one guard with a value y -- an input or a state variable -- and strengthen the other guard with $\neg y$, thus giving (y should be different from x):

$$B0 \wedge y \rightarrow x\uparrow \ ,$$
$$B1 \wedge \neg y \rightarrow x\downarrow \ .$$

The specification thus obtained satisfies the rule of disjointness independently of the structure of B0 and B1 .

In view of the above one probably will not be surprised when we present an element with three inputs s, r, g and output x with the following specification:

$$s \wedge g \rightarrow x\uparrow ,$$
$$\neg r \wedge \neg g \rightarrow x\downarrow .$$

The negation in front of r may seem somewhat arbitrary, but if we allow inverters to be used freely this negation is harmless. We call this element a "G-element" and within pictures, we draw it as in figure 4.0.

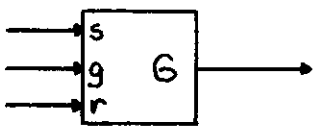

fig. 4.0: a G-element

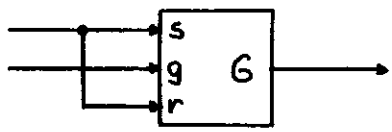

fig. 4.1: a C-element

The G-element is asymmetric in its inputs; therefore, in any drawing these inputs must be labelled. The G-element is a generalisation of the well-known C-element: a C-element is obtained by connecting the s and r inputs (See figure 4.1). This connection, however, must be considered as an internal fork. In section 5 we shall give an implementation of the G-element.

Secondly, we require all circuits to be free from, so-called, transition interference which is either transmission or computation interference (I do not very well know the difference, if any, between the latter two notions, hence the term transition interference): no transition may be sent along a wire before all previous transitions on the same wire have been "processed" by the receiving circuitry. In the case of delay-insensitive circuits this requirement gives always rise to some sort of feed-back, e.g. such as embodied in the four-phase hand-shaking protocol. However, in order to avoid hazards we also impose this condition onto the "internal operations" of the circuit. As a special case this condition gives rise to the following rule of monotonicity:

> No true guard may be falsified before the transition guarded by it has taken place.

Thirdly, we allow programs with commas. The constituents of a comma-ed program are called "processes". The parallellism thus introduced may be useful; after all, all elements constituting the circuit operate always in parallel. Now however we need a "Gries-Owicki like" rule:

> No true guard in one process may be falsified by any action of the other processes.

This rule is inspired by the observation that the guards in the programs may be viewed as assertions. Notice that this rule equals the rule of monotonicity as far as mutual internal operations are concerned. The rule of monotonicity, however, also pertains to inputs and to interference within a single process.

* * *

We recall from section 2 the specification of the circuit:

(ai↑;ao↑;ai↓;ao↓)* , (bi↑;bo↑;bi↓;bo↓)* ,
(ai↓;bo↑)* , (ao↑;bi↓)* .

We already know that the first two weavands -- thanks to the fact that the environment respects the protocol -- can be coded without problems as a program:

*[[ai];ao↑;[¬ai];ao↓] , *[[bi];bo↑;[¬bi];bo↓] .

We now take this program for our starting point. The remaining two weavands may be considered as additional restrictions on the traces allowed by this program -- this is in accordance with the conjunction-weave rule of trace theory -- . Additional restrictions may be implemented by strengthening the guards. Strengthening guards never destroys the correctness of the program but it may introduce the danger of deadlock. From section 3 we remember that at least one state variable is needed -- actually two, but we introduce them one at a time -- and thus we obtain our second approximation:

$*[[ai \wedge x];ao\uparrow;[\neg ai];ao\downarrow,x\downarrow]$,
$*[[bi \wedge \neg x];bo\uparrow;[\neg bi];bo\downarrow,x\uparrow]$.

Initially, x is true. The transitions of x have been placed so as to implement the additional restrictions; we now have: $ai\downarrow$ precedes $x\downarrow$ and $x\downarrow$ precedes $bo\uparrow$; hence, by transitivity: $ai\downarrow$ precedes $bo\uparrow$. Similarly, by symmetry, $bi\downarrow$ precedes $ao\uparrow$ but for the "first" $ao\uparrow$: here the initial value of x comes in. If we derive a specification for (an element realising) x from this program then we obtain either

$\neg ai \rightarrow x\downarrow$,
$\neg bi \rightarrow x\uparrow$, or, strengthening the guards a little,

$\neg ai \wedge ao \rightarrow x\downarrow$,
$\neg bi \wedge bo \rightarrow x\uparrow$.

In both cases the rule of disjointness is not satisfied. Therefore, we apply our standard trick to achieve disjointness by introducing a second state variable y ; we now get:

$\neg ai \wedge y \rightarrow x\downarrow$,
$\neg bi \wedge \neg y \rightarrow x\uparrow$.

The guards in the program are strengthened accordingly. Where can we safely insert a transition $y\uparrow$ into the program? Well, $y\uparrow$ may falsify the guard of $x\uparrow$; this causes no conflict with the rule of monotonicity if $y\uparrow$ takes place in a state where x is true. Furthermore, $y\uparrow$ never falsifies the guard of $x\downarrow$. The transition $y\downarrow$ is dealt with symmetrically and so we arrive at our third approximation:

$*[[ai \wedge x];ao\uparrow,y\uparrow;[\neg ai \wedge x \wedge y];ao\downarrow,x\downarrow]$,
$*[[bi \wedge \neg x];bo\uparrow,y\downarrow;[\neg bi \wedge \neg x \wedge \neg y];bo\downarrow,x\uparrow]$.

On afterthought, the place of the transitions of y is not that surprising when we remember the structure of the transition counter

of section 3. A specification for y is:

$$ai \wedge x \rightarrow y\uparrow \ ,$$
$$bi \wedge \neg x \rightarrow y\downarrow \ .$$

Notice that the program thus obtained satisfies the Gries-Owicki rule but that it does not satisfy the rule of monotonicity: $ai\uparrow;ao\uparrow;y\uparrow;ai\downarrow;ao\downarrow;ai\uparrow$ is a valid trace but $ai\uparrow$ now falsifies the true guard $[\neg ai \wedge x \wedge y]$. As a consequence, the circuit may "miss the opportunity" to perform $x\downarrow$. This situation can be avoided by requiring $x\downarrow$ to occur before the next $ai\uparrow$: then, $ai\uparrow$ occurs in a state where the guard $[\neg ai \wedge x \wedge y]$ already is false. To achieve this we exploit the freedom to replace $ao\downarrow, x\downarrow$ by $x\downarrow;[\neg x];ao\downarrow$. The guard $[\neg x]$ is not an invariant of the other process however, therefore we weaken it to $[\neg x \vee \neg y]$, which is an invariant of the other process. Thus, we arrive at our fourth and final approximation:

$$*[[ai \wedge x];y\uparrow;[x \wedge y];ao\uparrow;[\neg ai \wedge x \wedge y];x\downarrow;[\neg x \vee \neg y];ao\downarrow] \ ,$$
$$*[[bi \wedge \neg x];y\downarrow;[\neg x \wedge \neg y];bo\uparrow;[\neg bi \wedge \neg x \wedge \neg y];x\uparrow;[x \vee y];bo\downarrow] \ .$$

The specifications of x and y have been given above; they indicate the use of G-elements. Initially, x is true and y is false. The specifications of ao and bo now follow immediately from the program; they satisfy the rule of disjointness:

$$x \wedge y \rightarrow ao\uparrow \ , \qquad \neg x \wedge \neg y \rightarrow bo\uparrow \ ,$$
$$\neg x \vee \neg y \rightarrow ao\downarrow \ , \quad \text{and:} \quad x \vee y \rightarrow bo\downarrow \ .$$

These specifications indicate an And-gate and a Nor-gate respectively. The circuit thus obtained is drawn in figure 4.2. The "fat dots" in this drawing denote internal forks. Furthermore, notice that a can be made active by addition of an inverter whereas b can be made active by removal of an inverter (i.e. two inverters).

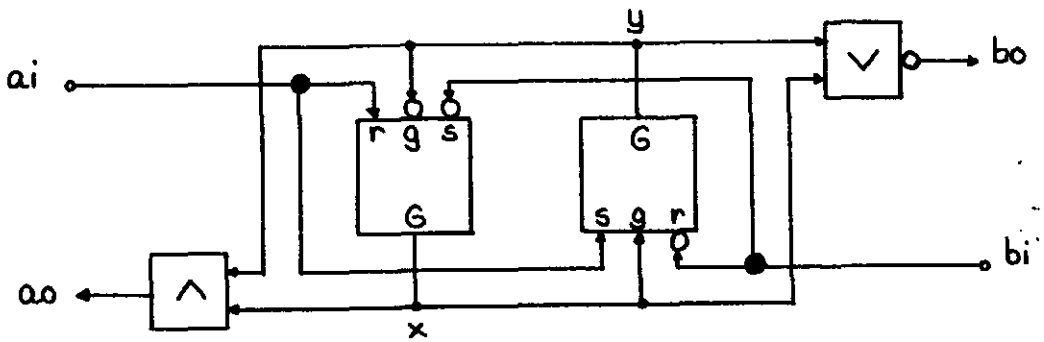

fig. 4.2: $SEM_1(a,b)$, a and b passive

5. More on G-elements

We consider elements with specification:

$B0 \rightarrow x\uparrow$,

$B1 \rightarrow x\downarrow$.

Remember that we require all elements to satisfy the rule of disjointness. Any such element can be implemented as follows.

Informally, x is true either because x becomes true or because x remains true. x becomes true when $B0$ is true and x remains true when $\neg B1 \wedge x$ holds. So, x is a fix-point of the following function f :

$$f.y \equiv B0 \vee (\neg B1 \wedge y) \ .$$

If we take into account the inputs of the element we observe that $B0$ and $B1$ are functions of the inputs; hence, f is a function having one more argument than the number of inputs. This function can be implemented in a standard way as a "combinatorial circuit". It is, however, essential that this circuit be free from the danger of hazards. The element can now be formed by connecting the y-input of the circuit with its output. The remaining inputs then are the inputs of the element. See figure 5.0, where the inputs are collectively labelled u .

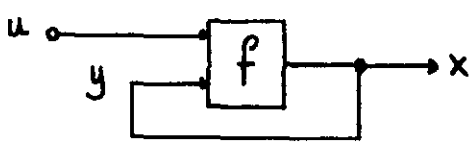

fig. 5.0: general element

The feed-back connection thus introduced must not introduce hazards. Therefore, the feed-back path must be at least as fast as the output

to the environment; in this respect the node in figure 5.0 may be interpreted as a "one-sided fork".

An interesting special case arises if $B0 \vee B1$ is true for all values of the inputs:

$$
\begin{aligned}
& B0 \vee (\neg B1 \wedge y) \\
= & \{ \text{ calculus } \} \\
& B0 \vee (\neg B0 \wedge \neg B1 \wedge y) \\
= & \{ \text{ de Morgan } \} \\
& B0 \vee (\neg (B0 \vee B1) \wedge y) \\
= & \{ B0 \vee B1 \text{ ; more calculus } \} \\
& B0 \ .
\end{aligned}
$$

Hence, the function f does not depend on y and the feed-back connection is superfluous. The resulting circuit is purely combinatorial; so, we might call such elements "combinatorial elements".

The specification of the G-element is:

$$
\begin{aligned}
& s \wedge g \rightarrow x\uparrow \ , \\
& \neg r \wedge \neg g \rightarrow x\downarrow \ .
\end{aligned}
$$

By substitution into the above definition of f we obtain:

$$
\begin{aligned}
& f.y \\
= & \{ \text{ definition of } f \ \} \\
& (s \wedge g) \vee (\neg (\neg r \wedge \neg g) \wedge y) \\
= & \{ \text{ de Morgan and distribution } \} \\
& (s \wedge g) \vee (r \wedge y) \vee (g \wedge y) \\
= & \{ \text{ change of order } \} \\
& (s \wedge g) \vee (g \wedge y) \vee (y \wedge r) \\
= & \{ \text{ calculus } \} \\
& (r \vee g) \wedge (g \vee y) \wedge (y \vee s) \ .
\end{aligned}
$$

The latter two forms are of interest: they give rise to two possible realisations of the G-element using And-gates and Or-gates

only. See figure 5.1. These two realisations are dual realisations with respect to negation; apparently, the G-element is invariant under negation of all wires provided the s and r inputs are interchanged. See also figure 5.2.

Lemma: Any combinatorial circuit consisting of And-gates and Or-gates only, is free from the danger of hazards.
proof: (omitted; hint: by structural induction).
(end of lemma)

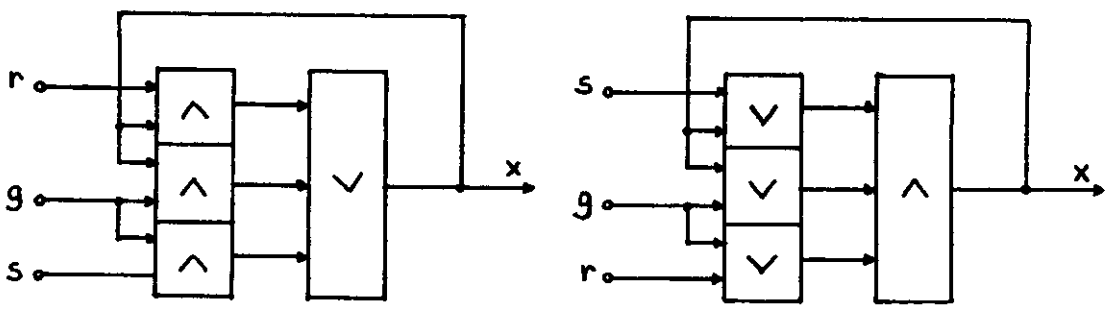

fig. 5.1: realisations of G-elements

Identification of the s and r inputs yields the well-known C-element; the resulting realisation is probably well-known too. The function f above then is what is sometimes called the "majority function". Finally, notice that any non-combinatorial element exhibits some form of storage: it has two states. Hence, in any practical realisation provisions must be made for the initialisation of the element's state.

fig. 5.2: duality property of G-elements.

6. In retrospect

One of the things I disliked very much in the current approach to circuit design was the notion of internal forks. As we have seen the use of internal forks seems unavoidable.every now and then. Let us, in order to illustrate this unavoidability, pay some more attention to the transition detector of section 3. See figure 6.0 for the circuit obtained by completing the design given in section 3. I use this example because of its relative simplicity. The names x and y in figure 6.0 correspond to the state variables introduced in section 3.

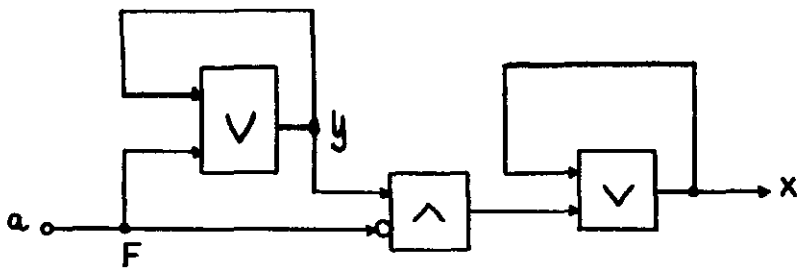

fig. 6.0: circuit for $a\uparrow;a\downarrow;x\uparrow;(a\uparrow;a\downarrow)*$

The node labelled F in figure 6.0 must be considered a (one-sided) fork, otherwise the circuit is incorrect. Since I am tempted to consider this circuit as the prototype of transition-sensitive circuits and since I see no way to avoid the fork even in this simple case, I am tempted to conclude that in the current approach forks are essential elements. Whether this is a defect of the theory or a consequence of the physical properties of delay-insensitive circuits is a question still to be answered. Unfortunately, the present theory does not cater for these forks; for a given circuit the internal forks are only identified by inspection, on afterthought so to speak.

A more pleasant observation is the following one. With exception of figure 1.0 all figures in this note only serve as illustrations; for someone -- like me -- who traditionally is used to reason about circuits in terms of pictures this is a substantial step forward. The implementation of a trace structure now seems to consist

of a number of steps:

- transformation of the trace structure into a trace structure specifying the circuit in terms of transitions. The greatest difficulty here seems to be the avoidance of overspecification. The distinction between active and passive seems to be of minor importance here. Actually, Asia van de Mortel-Froncak pointed out to me that when one derives a circuit for $SEM_1(a,b)$ with b active, the same circuit, but for one inverter, is obtained as in section 4.
- transformation of the expanded trace structure into a "program". This is difficult because of the transition from transitions to boolean values. Probably, even more strict rules are needed than the ones used in this note; it wouldn't surprise me at all when a Gries-Owicki like way of proving the correctness of the programs would turn out to be very effective.
- transformation of the program into a collection of specifications, one for each output and one for state variable, of elements. It is not clear yet whether this step can be completely separated from the previous one. A problem here is that all elements always operate concurrently; hence, we need a way to "get rid of the semicolons" in the program.

7. Acknowledgements and references

The activities leading to this report have been triggered by my desire to actually construct a $SEM_1(a,b)$ according to Anne Kaldewaij's circuit and my simultaneous desire to understand what I was constructing. The asymmetry mentioned in the introduction manifested itself in the prototype in such an annoying way that I set out to look for symmetric solutions. Therefore, thanks are due to Anne Kaldewaij for writing his thesis the way he did.

Furthermore, thanks are due the other members of the Kleine Club for providing an opportunity to experiment with various immature ideas, to Asia van de Mortel-Froncak for lending me her patient ear and for her helpful suggestions, and,finally, to Martin Rem for his encouragement.

references:

[0] Anne Kaldewaij: A formalism for concurrent processes.
dissertation, Eindhoven University of Technology, 1986.

[1] Alain J. Martin: The design of a self-timed circuit for distributed mutual exclusion.
Proceedings 1985 Chapel Hill Conference on VLSI, ed. H. Fuchs.
Computer Science Press, Rockville, 1985.

(1986.4.21, revisited 1986.9.26)

COMPUTING SCIENCE NOTES

In this series appeared

Nr.	Author(s)	Title
85/01	R.H. Mak	The Formal Specification and Derivation of CMOS-circuits
85/02	W.M.C.J. van Overveld	On arithmetic operations with M-out-of-N-codes
85/03	W.J.M. Lemmens	Use of a Computer for Evaluation of Flow Films
85/04	T. Verhoeff H.M.J.L. Schols	Delay insensitive Directed Trace Structures Satisfy the Foam Rubber Wrapper Postulate
86/01	R. Koymans	Specifying Message Passing and Real-time Systems
86/02	G.A. Bussing K.M. van Hee M. Voorhoeve	ELISA, A Language for Formal Specifications of Information Systems
86/03	Rob Hoogerwoord	Some reflections on the implementation of trace structures