



Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

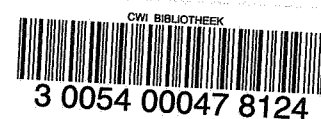
J.C. Ebergen

A technique to design delay-insensitive VLSI circuits

Computer Science/Department of Algorithmics & Architecture

Report CS-R8622

June



Bibliotheek
Centrum voor Wiskunde en Informatica
Amsterdam

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

A Technique to Design Delay-Insensitive VLSI Circuits

Jo C. Ebergen

Centre for Mathematics & Computer Science,
Department of Algorithmics and Architecture,
P.O. Box 4079, 1009 AB Amsterdam,
The Netherlands

A technique for a hierarchical design of delay-insensitive circuits is presented. The techniques are developed by means of the trace-theory formalism. The design consists of the formulation of a specification and its decomposition into basic elements. Parallelism is allowed in a specification. The notion of delay-insensitive circuit is formalized. Three examples are given to illustrate the technique.

1980 Mathematics Subject Classification: 68B10, 68D37, 68FXX, 94C99.

CR Categories: B.6.1, B.7.1, F.1.1.

Keywords & Phrases: delay-insensitive circuit, VLSI design, parallelism, trace semantics, specification, decomposition.

69B61
69B71
69F11

1. INTRODUCTION

The purpose of this paper is to present some techniques to design delay-insensitive circuits and to illustrate these techniques by some examples.

The reason to design *delay-insensitive* circuits is the avoidance of timing problems. These problems are created by, for example, delays in wires and switches, the glitch phenomenon [1], and scaling [9]. By designing delay-insensitive circuits one obtains a separation of timing concerns and functional correctness concerns, which simplifies the design of a VLSI circuit. Moreover, delay-insensitive circuits tend to be faster.

A circuit is formally described as a cooperation between a number of communicating components instead of a sequential finite-state machine as in classical switching theory. Accordingly, we allow parallelism in the formal descriptions of circuits. The design techniques are based on a formalism, trace theory, in which circuits can be specified adequately and in which one can perform calculations to obtain a decomposition of a specification into specifications of smaller circuits. The decomposition is such that the connection of the circuits specified satisfies the original specification irrespective of delays in connection wires. Thus the design of a circuit is separated into two parts: the decomposition of a specification into simpler constituents, and the physical realization of the basic constituents. It is only at the physical realization level that timing and other physical concerns become relevant. At the decomposition level the designer deals with the structural complexity of his design. In order to bridle this complexity, the formalism allows for a hierarchical decomposition method. In this paper we deal only with specifications and their decomposition and not

Report CS-R8622

Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

with physical realizations.

Related work in the design of delay-insensitive circuits, where different techniques and sometimes different formalisms are employed, can be found in [5, 6, 7, 9, 12, 13].

In Section 2 the trace-theory formalism is briefly introduced and it is shown how circuits can be specified. In Section 3 we give a formalization of decomposition and of delay-insensitive specifications. In Section 4 some design techniques are illustrated in the design of three circuits: a Quick Return Linkage, a 4-counter, and a token ring. We conclude with some remarks and conjectures.

2. SPECIFYING CIRCUITS

In this section we briefly present an informal account of trace theory. For a further introduction the reader is referred to [7, 13]. Trace theory also bears some resemblance to Hoare's CSP [3]. A trace structure R is a pair $\langle A, X \rangle$, where A is a finite set of symbols and $X \subseteq A^*$, in which A^* is the set of all finite sequences over A . The elements of A^* are called traces. A is called the *alphabet* of R , denoted by $\mathbf{a}R$ and X is called the *trace set* of R , denoted by $\mathbf{t}R$. The empty trace is denoted by ϵ . We use the following operators to construct trace structures:

$$\begin{aligned} R;S &= \langle \mathbf{a}R \cup \mathbf{a}S, \mathbf{t}R \mathbf{t}S \rangle \\ R|S &= \langle \mathbf{a}R \cup \mathbf{a}S, \mathbf{t}R \cup \mathbf{t}S \rangle \\ [R] &= \langle \mathbf{a}R, (\mathbf{t}R)^* \rangle \\ R||S &= \langle \mathbf{a}R \cup \mathbf{a}S, \{t \in (\mathbf{a}R \cup \mathbf{a}S)^* \mid \# \mathbf{a}R \in \mathbf{t}R \wedge \# \mathbf{a}S \in \mathbf{t}S\} \rangle \\ \text{pref}R &= \langle \mathbf{a}R, \{t \mid \exists s :: ts \in \mathbf{t}R\} \rangle \\ R \uparrow A &= \langle \mathbf{a}R \cap A, \{\# A \mid t \in \mathbf{t}R\} \rangle, \end{aligned}$$

where $\# A$ is the trace t projected on A , i.e., the trace t from which all symbols not in A have been removed. The above operations are called concatenation, union, repetition, weaving, taking the prefix-closure, and projection respectively. With respect to the priority of the binary operators we adopt that weaving has highest priority, then concatenation followed by union. The abbreviation R^2 is used for $R;R$.

A *directed* trace structure is a triple $\langle A, B, X \rangle$, such that $\langle A \cup B, X \rangle$ is a trace structure. A is called the input alphabet of R , also denoted by $\mathbf{i}R$; B is called the output alphabet of R , also denoted by $\mathbf{o}R$. We have $\mathbf{a}R = \mathbf{i}R \cup \mathbf{o}R$. For brevity's sake we adopt the convention that when the characters $a?$, $a!$, or a appear in the context of one of the above operators the trace structures $\langle \{a\}, \emptyset, \{a\} \rangle$, $\langle \emptyset, \{a\}, \{a\} \rangle$, or $\langle \{a\}, \{a\}, \{a\} \rangle$ respectively are meant. The above operations are defined on directed trace structures in a similar way, i.e., the construction of the trace set remains the same and the construction of the alphabet is split into the analogous construction of the input alphabet and the output alphabet. For example, $R;S = \langle \mathbf{i}R \cup \mathbf{i}S, \mathbf{o}R \cup \mathbf{o}S, \mathbf{t}R \mathbf{t}S \rangle$ for directed R and S .

We specify a circuit by means of a prefix-closed, non-empty, directed trace structure R for which $\mathbf{i}R \cap \mathbf{o}R = \emptyset$. In fig. 1 a number of specifications for circuits are given. For each circuit its name, a specification in the form of an expression or a so-called *command*, and its schematic is given. For some circuits an alternative command is given with the same trace structure.

All specifications in fig. 1 are given starting in a certain initial state. An other specification of the same circuit, but starting in a different initial state, can be represented schematically by putting inverters in terminal wires. For example the C-element started in a state where the first occurrence of input b has already occurred is specified and depicted as in fig. 2.



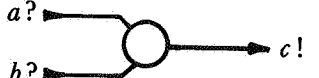



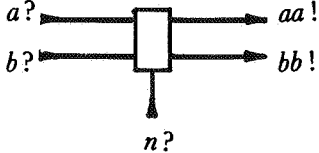
Name	Specification	Schematic
Wire	$W(a, b) = \text{pref}[a?; b!]$	
Fork	$\text{pref}[a?; b! \parallel c!]$ $= \text{pref}[a?; b!] \parallel \text{pref}[a?; c!]$	
C-element	$\text{pref}[a? \parallel b?; c!]$	
C-element with 2 replicated inputs	$\text{pref}[(a?; e!)^2 \mid (b?; d!)^2$ $\mid (a?; e! \parallel c!)^2 \parallel (b?; d! \parallel c!)^2]$	
Exor	$\text{pref}[(a? \mid b?); c!]$	
Toggle	$\text{pref}[a?; b!; a?; c!]$	
Sequentializer	$\text{pref}[a?; aa!] \parallel \text{pref}[b?; bb!]$ $\parallel \text{pref}[aa!; n? \mid bb!; n?]$	

fig. 1


C-element in a different initial state	$\text{pref}(a?; [c!; a? \parallel b?])$ $= \text{pref}[a?; c!] \parallel \text{pref}[c!; b?]$	
---	---	---

fig. 2

A specification R is operationally interpreted as follows. With each symbol in aR a terminal of the circuit is associated. These points form the *boundary* aR between the circuit and its environment. Initially the voltage levels at these points are 0. Transitions in these voltage levels, i.e., changes from 0 to 1 or from 1 to 0, can be caused by the environment, if the symbol is an input, or by the circuit, if the symbol is an output. Environment and circuit can engage in many behaviors by causing transitions in the patterns specified. The specification is read as a prescription for the behavior of *both* circuit *and* environment at their boundary. For example the specification for the Fork prescribes that the environment starts with causing a transition in the point associated with a . Then the circuit must cause transitions at b and c , which may occur in arbitrary order. The environment gives a new transition at a only after it has received both transitions at b and c . Subsequently, the circuit causes transitions at b and c again, etc.. Thus a game is played between environment and circuit where each in turn must make a move (a transition) according to the rules as laid down in the specification. For reasons of simplicity we deal with non-terminating components only in this paper.

To maintain a clear distinction between formalism and physical realization we talk about components instead of circuits, where component should be understood as a kind of automaton and circuit as a physical realization.

3. DECOMPOSITION AND DELAY-INSENSITIVITY

We begin this section by defining decomposition of a specification. The motivation behind the definition is that if we say ' R can be decomposed into S and T ', then we may interpret this by 'the circuit as specified in R can be a connection of the circuits as specified in S and T '.

Notice that we talk about *decomposing* a specification and not composing two, or more, specifications to obtain a new one. As explained in the previous section, a specification R gives a prescription of a component in a prescribed environment. We take this environment into account when we look for a connection of components that satisfies the behavior of the prescribed component in R . If we would consider only S and T in a composition method, then we exclude the effect of the environment, as prescribed in R , in which the components in S and T are supposed to operate. Therefore we consider not only S and T , but also R to define our decomposition method.

Consider the cooperation of the components in S and T , and of the environment in R . The environment in R is also prescribed as the component in \bar{R} , i.e., the *reflection* of R . The reflection of R is obtained by making the outputs of R inputs, and making the inputs of R outputs, e.g., if $R = \text{pref}[a?; b!]$, then $\bar{R} = \text{pref}[a!; b?]$, $i\bar{R} = oR$, and $o\bar{R} = iR$. The cooperation of the components is consequently described by the weave $W = \bar{R} || S || T$. This weave W can be interpreted as the representation of all behaviors of the connection of the components in \bar{R} , S , and T at the points aW such that the respective boundary prescriptions are satisfied.

In order for S and T to be a decomposition of R some restrictions have to be satisfied and a phenomenon, called *interference*, must not occur. The restrictions are defined by means of the alphabets. Interference is defined below by means of the weave W . The first restriction is that dangling inputs or outputs must not occur, i.e.,

$$(1) \quad o\bar{R} \cup oS \cup oT = i\bar{R} \cup iS \cup iT.$$

In order to prevent 'shortcircuiting', the second restriction is that no outputs are connected with each other, i.e.,

(2) $\mathbf{o}\bar{R} \cap \mathbf{o}S = \emptyset$, $\mathbf{o}\bar{R} \cap \mathbf{o}T = \emptyset$, and $\mathbf{o}S \cap \mathbf{o}T = \emptyset$.

The phenomenon interference can be described as the violation of a boundary prescription. More formally, suppose there exists a trace $t \in \mathbf{t}W$ such that after this trace a component, as prescribed in V say, where $V \in \{\bar{R}, S, T\}$, is enabled to generate an output b say. The generation of b , however, is not in accordance with the boundary prescriptions of the other components in V , i.e.,

$$t \in \mathbf{t}W \wedge b \in \mathbf{o}V \wedge tb \uparrow \mathbf{a}V \in \mathbf{t}V \wedge tb \notin \mathbf{t}W.$$

We speak then of a boundary violation, or danger of interference. In the case of a boundary violation for a Wire, operationally seen, more than one transition is propagating on a wire, which can cause hazardous behavior, and must therefore be prevented. Interference for a Wire is also called transmission interference, otherwise we speak of computation interference [13].

Finally, the cooperation of the components in \bar{R} , S , and T must behave at the boundary $\mathbf{a}R$ as prescribed in R , i.e., $\mathbf{t}W \uparrow \mathbf{a}R = \mathbf{t}R$. We can now give

Definition 3.1 We say that R can be decomposed into S and T if the restrictions (1) and (2) hold, the weave of \bar{R} , S , and T is free of interference, and $\mathbf{t}W \uparrow \mathbf{a}R = \mathbf{t}R$. \square

Note. The above definition does not include the requirement that other phenomena such as deadlock and unbounded internal chatter (or livelock [4]) must be absent. Although these phenomena are essential for an acceptable definition of decomposition, they are not dealt with in detail in this paper. In the last section a few words are devoted to these phenomena. (*End of Note.*)

The above definition can, in an obvious way, be extended to decompositions into more than two trace structures. Moreover, this definition allows for an hierarchical decomposition method, since we have

Theorem 3.2 If R can be decomposed into S_0 and S_1 , and S_0 can be decomposed into S_2 and S_3 , then R can be decomposed into S_1 , S_2 , and S_3 . \square

The boundary at which the behavior of component and environment is prescribed is considered a fixed boundary. From a physical realization point of view, connecting circuits with fixed boundaries is generally impossible; usually one uses connection wires. Instead of incorporating connection wires in a each decomposition, we rather consider specifications that can be used for circuits with flexible boundaries. That is, we use specifications that remain invariant if the terminals of the prescribed components are extended by Wires. Since, physically spoken, connection wires introduce delays, specifications for such circuits with flexible boundaries are often called delay-insensitive. In order to abstract from a too physical interpretation in our formalism we say that a specification has property DI, or shortly, is DI. We define this property more formally. Let R' be the specification R where each symbol b is renamed by b' . Let *Wires* be the (disjoint) collection of wires $W(b', b)$ and $W(b, b')$ for $b \in \mathbf{i}R$ and $b \in \mathbf{o}R$ respectively. The property DI for a specification is then defined by

Definition 3.3 A specification R is DI if and only if R' can be decomposed into R and *Wires*. \square

Theorem 3.2 and definition 3.3 imply that in any decomposition in which only DI specifications are used, also connection Wires may be incorporated without influencing the correctness of the decomposition. This means that in the corresponding correct realization, delays incurred in connection wires do not influence the correct functioning of the circuit.

Charles Molnar was one of the first who tried to formalize the notion of delay-insensitivity by means of the so-called Foam Rubber Wrapper principle [6]. Jan Tijmen Udding has defined a number of rules in [12] in order to conclude that a specification satisfies the Foam Rubber Wrapper principle. It turns out that definition 3.3 is equivalent to these rules. In [8] a formalization of the Foam Rubber Wrapper principle is given which is also equivalent to definition 3.3. We do not elaborate on the recognition of the DI property any further in this article. All specifications occurring in this article are DI.

In the next section we illustrate a technique to decompose DI specifications by stepwise refinement into a number of basic elements. As a starting point we take a DI command satisfying a certain syntax. In a sense the construction of the command determines the construction of the decomposition. At the end of a stepwise decomposition the complete decomposition can be given by application of the substitution theorem 3.2. The technique is such that in each step the decomposition is correct by construction. We do not prove this here. Finally, we remark that the techniques presented are neither complete nor generally applicable. They only serve as an indication of how a general method may proceed.

4. EXAMPLES

Example 4.1. The specification of the Quick Return Linkage (*QRL*) reads as follows

$$QRL = \text{pref}(a?; [b!; c?; d!; a?; (b!; c?) || (d!; a?)]) .$$

This specification is DI. A schematic for the *QRL* is depicted in figure 3.



fig. 3

In the specification *QRL* we can distinguish between the odd and even occurrences of a symbol. We denote these occurrences by $b1$ and $b0$ respectively for the symbol b , say. (Operationally seen, we make distinctions between high-going transitions and low-going transitions in a terminal.) The specification then reads

$$QRL' = \text{pref}(a1?; [b1!; c1?; d1!; a0?; (b0!; c0?) || (d0!; a1?)]) .$$

Note. The *QRL* serves as a link between two components to signal initiation and completion of actions. The initiation of an action is sent by the high-going transitions $a1$, $b1$, $c1$, and $d1$. The completion is sent by the low-going transitions $a0$, $b0$, $c0$, and $d0$. This phase is also called the 'return-to-zero' phase. The component deserves its name because d may return to zero before b and c do. (*End of Note.*)

From the specifications in fig. 1, we deduce that the distinction between odd and even occurrences for an input can be accomplished by a Toggle. The abstraction from odd and even occurrences for an output can be made by an Exor; see also fig. 1. Consequently, if we are able to decompose *QRL'*, then, by the substitution theorem 3.2, the complete decomposition of *QRL* can be depicted as in fig. 4.

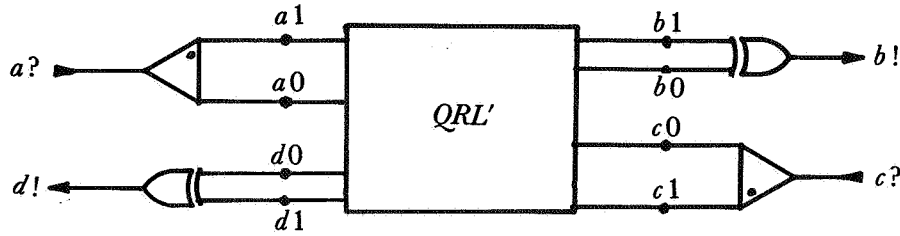


fig. 4

For the decomposition of QRL' we rewrite $QRL' \parallel \overline{QRL'}$ as a set of production rules, i.e.,

$$\text{pref}(a1?; [b1!; c1?; d1!; a0?; (b0!; c0?) \parallel (d0!; a1?)])$$

$$\parallel \overline{QRL'}$$

$$= \text{pref}(a1?; [b1!; c0? \parallel a1?]) \quad (1)$$

$$\parallel \text{pref}[c1?; d1!] \quad (2)$$

$$\parallel \text{pref}[a0?; b0!] \quad (3)$$

$$\parallel \text{pref}[a0?; d0!] \quad (4)$$

$$\parallel \overline{QRL'}$$

Above we gave for each output in QRL' its immediate preceding inputs that cause the production of this output. In line (1) we have given the production rule for $b1!$; in line (2) for $d1!$; in line (3) for $b0!$; and in line (4) for $d0!$. One could say that the semicolons between inputs and outputs in QRL' are realized by rules (1) thru (4); the other semicolons are realized by the environment, i.e., by $\overline{QRL'}$.

From line (1) thru (4) we can derive a decomposition for QRL' into a C-element, cf. line (1) and fig. 2.; a Wire, cf. line(2) and fig. 1.; and a Fork, cf. line (3) and (4) and fig. 1. A schematic of this decomposition is depicted in fig. 5.

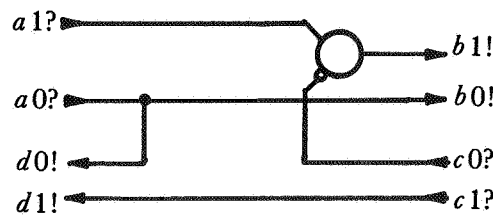


fig. 5

Example 4.2. A specification of the 4-counter reads as follows

$$C_4 = \text{pref}[a?; b; p!] \parallel \text{pref}[b; c] \parallel \text{pref}[c; d] \parallel \text{pref}[d; q!; e?] \uparrow \{a?, p!, q!, e?\}$$

Note. For the above counter, $a?$ and $e?$ denote the increment and the decrement; $p!$ and $q!$ are acknowledgements of the increment and decrement respectively. For every trace in C_4 , the number of increments minus the number of decrements is at least 0 and at most 4. Moreover, every trace with this property is also contained in C_4 . For the derivation of such an expression we refer to [7]. (*End of Note.*)

We first look at the command for C_4 without projection. In this command the so-called internal symbols b , c , and d occur. In order to obtain a DI command, we transform each internal symbol x into $x!$; $x'?$. This yields

$$C_4 = \text{pref}[a?; b!; b'?: p!] \parallel \text{pref}[b!; b'?: c!; c'?] \\ \parallel \text{pref}[c!; c'?: d!; d'?] \parallel \text{pref}[d!; d'?: q!; e?]$$

Notice that still $C_4 \uparrow \{a?, p!, e?, q!\} = C_4$. We have that C_4 can be decomposed into C_4 , $W(b, b')$, $W(c, c')$, and $W(d, d')$.

For the decomposition of C_4 , we rewrite $C_4 \parallel \overline{C_4}$ in the same way as in the previous example to obtain a production rule for each output in C_4 . We obtain

$$C_4 \parallel \overline{C_4} \\ = \text{pref}[a?; b!] \parallel \text{pref}[b!; c'?] \tag{1} \\ \parallel \text{pref}[b'?: p!] \tag{2} \\ \parallel \text{pref}[b'?: c!] \parallel \text{pref}[c!; d'?] \tag{3} \\ \parallel \text{pref}[c'?: d!] \parallel \text{pref}[d!; e?] \tag{4} \\ \parallel \text{pref}[d'?: q!] \tag{5} \\ \parallel \overline{C_4}$$

From this equation we can find a decomposition for C_4 , consisting of three C-elements, cf. lines (1), (3), and (4), for the outputs b , c , and d respectively; three Forks, cf. lines (2) and (3) for the input b' , lines (1) and (4) for the input c' , and lines (3) and (5) for the input d' . The complete decomposition is depicted in the schematic in fig. 6.

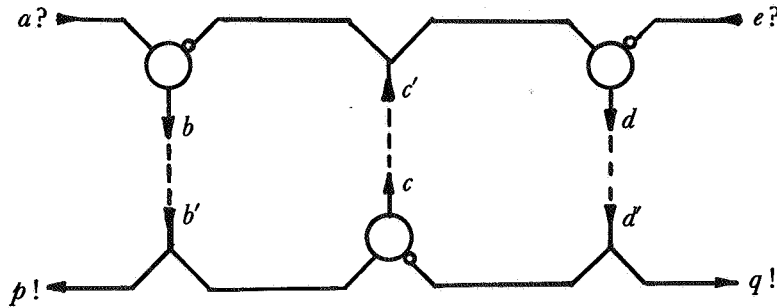


fig. 6

The circuit that we derived in this example is used in several delay-insensitive VLSI designs. For instance, it serves as a control structure for delay-insensitive FIFOs or delay-insensitive pipeline structures [10, 11].

Example 4.3. In this example we specify a token ring and indicate briefly how it can be decomposed into basic elements. For a more detailed derivation we refer to [3]. A different design for a similar token ring, though using other techniques, can be found in [5].

All processes are connected in a ring in which a so-called token is travelling from process to process. Only when a process has got hold of the token may it enter its critical section. After leaving the critical section it releases the token in the ring again. Since there is at most one token in the ring, there is at most one process engaged in its critical section, and,

accordingly, mutual exclusion between the critical sections is guaranteed.

A process communicates with the token ring through the ring interface, which is specified by the command

$$R = \text{pref}[a1?; p1!; a0?; p0!] \\ \parallel \text{pref}[b?; (q! | p1!; a0?; q!)]$$

With each symbol the following meaning can be associated.

- $a1?$ request of process to enter its critical section;
- $p1!$ grant to enter critical section;
- $a0?$ exit of process from critical section;
- $p0!$ acknowledgement of leave;
- $b?$ receipt of token from left neighbour;
- $q!$ release of token to right neighbour.

First, we can derive that R can be decomposed into the components $\text{pref}[a0?; p0!]$, $S0$, $T0$, and $S1$, where

$$S0 = \text{pref}[a1?; aa! | \text{pref}[b?; bb! | \text{pref}[aa!; n? | bb!; n?]. \\ T0 = \text{pref}[aa? || s0?; t0! | bb? || s1?; t1! | bb? || s0?; t2!]$$

and

$$S1 = \text{pref}(s0!; [t0?; n! || s1! | t1?; n! || s0! | t2?; n! || s0!]) \\ \parallel \text{pref}[t1?; p1! | t2?; q! | a0?; q!]$$

Here, $S0$ is a basic element, the so-called 'Sequentializer'.

The next step in the hierarchical decomposition is the decomposition of $S1$ and $T0$. The component $S1$ can be readily decomposed into Exors. This decomposition is similar to the construction of the Or-plane in a PLA. $T0$ is decomposed into a converter that converts between 2-cycle signalling and 4-cycle signalling ([9]) and the 4-cycle version of $T0$, denoted by $T0'$ and given by

$$T0' = \text{pref}[(aa? || s0?; t0')^2 | (bb? || s1?; t1')^2 | (bb? || s0?; t2')^2].$$

Finally, $T0'$ can be decomposed into C-elements with and without replicated inputs. The complete decomposition is depicted in fig. 7.

5. CONCLUDING REMARKS

In the above we have presented a short introduction to the specification of circuits, their decomposition, and the formalization of the notion 'delay-insensitive circuit'. We also illustrated by examples some techniques which yield a correct-by-construction decomposition of a circuit.

Although the techniques were applied to specific and simple examples, also other and more complicated circuits have been designed in a similar fashion. These designs include several token-ring configurations, a termination-detection configuration ([2]), sequence detectors, and traffic-light controllers.

The essence of the approach to circuit design presented here, and the main difference with classical switching theory, is the formal description of a circuit as a cooperation of communicating components. The underlying theory enables us to reason about the phenomenon of interference, by which the notion of delay-insensitivity can be formalized, and to develop techniques to design circuits in which interference does not occur. Moreover, parallelism is allowed in the design of these circuits.

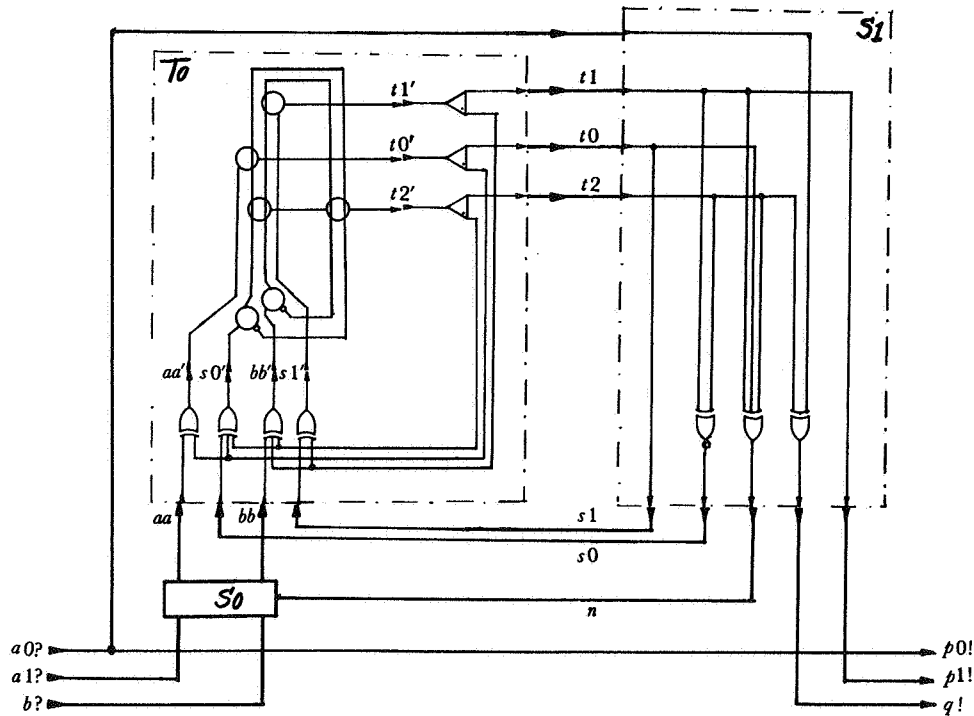


fig. 7

Although the approach presented has nice perspectives, there are still unsolved problems; problems that are introduced by non-determinism and parallelism. They have to do with progress requirements of a decomposition, i.e., *deadlock* and *unbounded internal chatter* (or livelock [4]). Let me explain these phenomena informally by way of example. In fig. 8 a decomposition of a Wire, $W(a, b)$, is given.

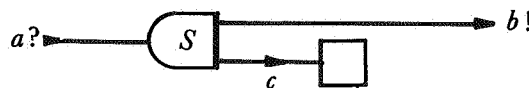


fig. 8

The specification of the so-called Select component reads $\text{pref}[a?;(b!|c!)]$, for the Sink we have $\text{pref}c?$. In this decomposition there is absence of interference and every trace in $W(a, b)$ can also be generated by this decomposition. However, after having received an arbitrary number of inputs a , it can stop generating outputs b . If we consider our Wire as a non-terminating component, then this is not a correct decomposition.

In fig. 9 an other decomposition of the Wire $W(a, b)$ is given. Also here there is absence of interference and every trace in $W(a, b)$ can be generated by the decomposition. However, after the receipt of an input a , it can take an unbounded sequence of internal symbols c before an output b occurs. If we do not allow an unbounded number of internal symbols to occur after every external symbol, then the above decomposition is also incorrect. Absence of deadlock and livelock are not required in our definition decomposition in this

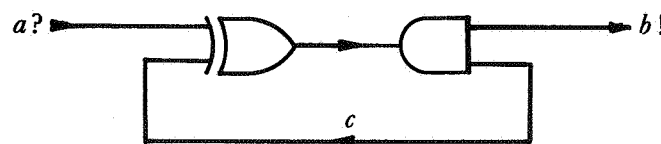


fig. 9

paper, though any decomposition method should guarantee absence of these phenomena. The development of a decomposition method such that absence of these phenomena is guaranteed is still a topic of research. The above designs in example 4.1, 4.2, and 4.3 have absence of deadlock and livelock.

Other interesting topics of research are the following. Can the above design techniques be generalized to a decomposition method that can be applied to any DI command? Does there exist a finite base of primitive elements into which any DI command can be decomposed in a constructive way? Moreover, can we say anything about the complexity of such a method, e.g., with respect to the number of primitive elements needed?

Recent research at CWI has given sufficient confidence to conjecture that any DI specification can be decomposed constructively into a finite basis of primitive elements. Such a set of primitive elements is {Sequentializer, Fork, C-element with and without replicated inputs, Toggle, Exor}. (The Sequentializer can also be replaced by the Arbiter.) Furthermore, it is conjectured that the number of primitive elements needed in a decomposition of a command satisfying a certain syntax is of the order of the length of that command.

Acknowledgements

Acknowledgements are due to the members of the Eindhoven VLSI Club and Lambert Meertens.

References

- [1] Chaney, T.J., Molnar, C.E., Anomalous Behavior in Synchronizer and Arbiter Circuits, *IEEE Transactions on Computers*, vol. C-22, no.4, April 1973, pp. 421-422.
- [2] Dijkstra, E.W., Feijen, W.H.J., van Gasteren, A.J.M., Derivation of a Termination Detection Algorithm for Distributed Computations, *Information Processing Letters*, vol. 16, 1983, pp. 217-219.
- [3] Ebergen, Jo C., A Circuit Design for a Token Ring: an Example in the Derivation of Delay-insensitive VLSI Circuits, C.W.I. report, to appear, 1986.
- [4] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice-Hall International, 1985.
- [5] Martin, A.J., The Design of a Self-timed Circuit for Distributed Mutual Exclusion, in: Fuchs, H., *1985 Chapel Hill Conference on Very Large Scale Integration*, Computer Science Press, 1985.
- [6] Molnar, C.E., Fang, T.-P., Rosenberger, F.U., Synthesis of Delay-insensitive Modules, in: Fuchs, H., *1985 Chapel Hill Conference on Very Large Scale Integration*, Computer Science Press, 1985.

- [7] Rem, M., Concurrent Computations and VLSI Design, in: Broy, M., *Control Flow and Data Flow: Concepts of Distributed Programming*, Springer-Verlag, 1985.
- [8] Schols, H., A Formalization of the Foam Rubber Wrapper Principle, Master's Thesis, Eindhoven University of Technology, 1985.
- [9] Seitz, C.L., System Timing, Chapter 7 in Mead & Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.
- [10] Seitz, C.L., Personal Communication.
- [11] Sproull, R.F, FIFO Controls for Four-Phase Storage Elements, Position paper for Workshop on Computational Models and Realizations, Institute for Biomedical Computing, Washington University, St. Louis, 1985.
- [12] Udding, J.T., Classification and Composition of Delay-insensitive Circuits, Ph. D. Thesis, University of Technology Eindhoven, 1984.
- [13] van de Snepscheut, J., Trace Theory and VLSI Design, Ph. D. Thesis, University of Technology Eindhoven, 1982.