EINDHOVEN UNIVERSITY OF TECHNOLOGY

A formal approach to designing delay-insensitive circuits

Citation for published version (APA):
Ebergen, J. C. (1988). *A formal approach to designing delay-insensitive circuits*. (Computing science notes; Vol. 8810). Technische Universiteit Eindhoven.

Document status and date:
Published: 01/01/1988

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

8810

ARD
01
CSN
8810

Eindhoven University of Technology

Department of Mathematics and Computing Science
Computing Science Section

Computing Science Notes

A Formal Approach to Designing Delay-Insensitive Circuits

by

Jo C. Ebergen

88/10

A Formal Approach to Designing Delay-Insensitive Circuits

by

Jo C. Ebergen

88/10

May 1988

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science of Eindhoven University of Technology.
Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review.
Copies of these notes are available from the author or the editor.

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB Eindhoven
The Netherlands
All rights reserved
editor: F.A.J. van Neerven

A Formal Approach to Designing Delay-Insensitive Circuits

Jo C. Ebergen
Dept. of Computing Science and Mathematics [1]
Eindhoven University of Technology
P.O. Box 513
5600 MB Eindhoven

A method for designing delay-insensitive circuits is presented based on a simple formalism. The communication behavior of a circuit with its environment is specified by a regular expression-like program. Based on formal manipulations this program is then transformed into a delay-insensitive connection of basic elements realizing the specified circuit. The notion of delay-insensitivity is concisely formalized.

1. Introduction

In 1938 Claude E. Shannon wrote his seminal article [23] entitled 'A Symbolic Analysis of Relay and Switching Circuits'. He demonstrated that Boolean algebra could be used elegantly in the design of switching circuits. The idea was to specify a circuit by a set of Boolean equations, to manipulate these equations by means of a calculus, and to realize this specification by a connection of basic elements. The result was that only a few basic elements, or even one element such as the 2-input NAND gate, suffice to synthesize any switching function specified by a set of Boolean equations. Shannon's idea proved to be very fertile and out of it grew a complete theory, called switching theory, which is used by most circuit designers nowadays.

The purpose of this paper is to present a formal approach to designing VLSI circuits, in particular *delay-insensitive circuits*. A delay-insensitive circuit can be interpreted as a circuit whose functional operation is insensitive to delays in basic elements or connection wires. The principal idea of our approach is similar to that in Shannon's article: to design a circuit as a connection of basic elements and to construct this connection from a specification with the aid of a formalism. We specify circuits by means of programs expressing orderings of events instead of logic functions. We then manipulate these programs by means of a calculus into a set of programs that correspond to basic elements. The connection of these elements forms a realization of the desired circuit.

The formal techniques and examples presented here form an extract from [5]. There, a method for constructing delay-insensitive circuits is developed that amounts to translating programs satisfying a certain syntax. The result of such a translation is a delay-insensitive connection of elements chosen from a finite set of basic elements. Moreover, this translation

1. The research reported in this article was carried out while the author was working at CWI (Centre for Mathematics and Computer Science) in Amsterdam.

has the property that the number of basic elements in the connection is proportional to the length of the program.

In this paper we give a short introduction to the formalism presented in [5] and illustrate this by means of some examples. The approach is briefly described as follows. An abstraction of a circuit is called a *component*; components are specified by programs written in a notation based on *trace theory*. Trace theory was inspired by Hoare's CSP [6, 7] and developed by a number of people at the University of Technology in Eindhoven for reasoning about parallel computations [9, 18, 19, 24] and delay-insensitive circuits [10, 20, 21, 26, 27].

The programs are called *commands* and can be considered as an extension of the notation for regular expressions. Any component represented by a command can also be represented by a regular expression, i.e. it is a *regular* component. The notation for commands, however, allows for a more concise representation of a component due to the additional programming primitives in this notation. These extra programming primitives include operations to express parallelism and projection (for introducing internal symbols).

Based on trace theory the concepts of *decomposition* and *DI decomposition* of a component are formalized. A decomposition of a component is intended to represent a realization of that component by means of a connection of other components such that the functional behavior of the connection is insensitive to delays in the components. Several theorems are presented that are helpful in finding decompositions of a component.

A DI decomposition represents a realization of a component by means of a connection of components such that the functional behavior of the connection is insensitive to delays in components *and* connection wires. In general, decomposition and DI decomposition are not equivalent. If, however, all constituting components are so-called *DI components*, then decomposition and DI decomposition are equivalent. Operationally speaking, a DI component represents a circuit for which the communication behavior between circuit and environment is insensitive to wire delays in those communications.

As examples we specify a modulo-3 counter and a token-ring interface by means of a command. Using the theorems presented, we then derive (DI) decompositions for these components into basic elements.

Before we discuss these examples and the underlying formalism, we describe some of the history of designing delay-insensitive circuits and some of the reasons why we would like to design delay-insensitive circuits.

2. Some History

Delay-insensitive circuits are a special type of circuits. We briefly describe their origins and how they are related to other types of circuits and design techniques. The most common distinction usually made between types of circuits is the distinction between *synchronous circuits* and *asynchronous circuits.*

Synchronous circuits are circuits that perform their (sequential) computations based on the successive pulses of the clock. From the time of the first computer designs many designers have chosen to build a computer with synchronous circuits. Alan Turing, one of those first computer designers, has motivated this choice as follows in [25]:

> We might say that the clock enables us to introduce a discreteness into time, so that time for some purposes can be regarded as a succession of instants instead of a continuous flow. A digital machine must essentially deal with discrete objects, and in the case of the ACE this is made possible by the use of a clock. All other digital computing machines except for human and other brains that I know of do the same. One can think up ways of avoiding it, but they are very awkward.

In the past fifty years many techniques for the design of synchronous circuits have been developed and are described by means of switching theory. The correctness of synchronous systems relies on the bounds of delays in elements and wires. The satisfaction of these delay requirements cannot be guaranteed under all circumstances, and for this reason problems can crop up in the design of synchronous systems. (Some of these problems are described in the next section.) In order to avoid these problems interest arose in the design of circuits without a clock. Such circuits have generally been called *asynchronous* circuits.

The design of asynchronous circuits has always been and still is a difficult subject. Several techniques for the design of such circuits have been developed (e.g. by Huffman) and are discussed in, for example, [11, 15, 28]. For special types of such circuits formalizations and other design techniques have been proposed and discussed. David E. Muller gave a rigorous formalization of a special type of circuits for which he coined the name *speed-independent* circuits. An account of this formalization is given in [16]. Informally speaking, speed-independent circuits are characterized as circuits that are insensitive to element delays.

From a design discipline that was developed as part of the Macromodules project [3, 4] at Washington University in St. Louis the concept of a special type of circuit evolved which was given the name *delay-insensitive* circuit, i.e. a circuit that is insensitive to both element delay and wire delay. It was realized that a proper formalization of this concept was needed in order to specify and design such circuits in a well-defined manner. A formalization of one of the central concepts in the design of delay-insensitive circuits, viz. that of the so-called 'Foam Rubber Wrapper' principle, was later given in [26].

Another name that is frequently used in the design of asynchronous circuits is *self-timed systems*. This name has been introduced by C. L. Seitz in [22] in order to describe a method of system design without making any reference to timing except in the design of the self-timed elements.

Recently, Alain Martin has proposed some interesting and promising design techniques for circuits of which the functional operation is unaffected by delays in elements [12, 13]. His techniques are based on the compilation of CSP-like programs into connections of basic elements. The techniques presented in [5] exhibit a similarity with the techniques applied by Alain Martin in the sense that they are both aimed at the translation of programs into connections of basic elements.

3. Why Delay-Insensitive Circuits ?

The reasons to design delay-insensitive systems are manifold. One reason why there has always been an interest in asynchronous systems is that synchronous systems tend to reflect a *worst-case* behavior, while asynchronous systems tend to reflect an *average-case* behavior.

A synchronous system is divided into several parts, each of which performs a specific computation. At a certain clock pulse, input data are sent to each of these parts and at the next clock pulse the output data, i.e. the results of the computations, are sampled and sent to the next parts. The correct operation of such an organization is established by making the clock period larger than the worst-case delay for any subcomputation. Accordingly, this worst-case behavior may be disadvantageous in comparison with the average-case behavior of asynchronous systems.

Another important reason for designing delay-insensitive systems is the so-called *glitch phenomenon.* A glitch is the occurrence of metastable behavior in circuits. Any computer circuit that has a number of stable states also has metastable states. When such a circuit gets into a metastable state, it can remain there for an indefinite period of time before it resolves into a stable state. For example, it may stay in the metastable state for a period larger than the clock period. Consequently, when a glitch occurs in a synchronous system, erroneous data may be sampled at the time of the clock pulses. In a delay-insensitive system it does not matter whether a glitch occurs: the computation is delayed until the metastable behavior has disappeared and the element has resolved into a stable state. One frequent cause for glitches are, for example, the asynchronous communications between independently clocked parts of a system.

The first mention of the glitch problem appears to date back to 1952 (cf. [1]). The first publication of experimental results of the glitch problem and a broad recognition of the fundamental nature of the problem came only after 1973 [2, 8] due to the pioneering work on this phenomenon at the Washington University in St. Louis.

A third reason is due to the effects of *scaling.* This phenomenon became prominent with the advent of integrated circuit technology. Because of the improvements of this technology, circuits could be made smaller and smaller. It turned out, however, that if all characteristic dimensions of a circuit are scaled down by a certain factor, including the clock period, delays in long wires do not scale down proportional to the clock period [14, 22]. As a consequence, some VLSI designs when scaled down may no longer work properly anymore, because delays for some computations have become larger than the clock period. Delay-insensitive systems do not have to suffer from this phenomenon if the basic elements are chosen small enough so that the effects of scaling are negligible with respect to the functional behavior of these elements ([24]) and the interconnections of these elements are delay-insensitive.

A fourth reason is the clear separation between functional and physical correctness concerns that can be applied in the design of delay-insensitive systems. The correctness of the behavior of basic elements is proved by means of physical principles only. The correctness of the behavior of connections of basic elements is proved by mathematical principles only. Thus, it is in the design of the basic elements only that considerations with respect to delays in wires play a role. In the design of a connection of basic elements no reference to delays in wires or elements is made. This does not hold for synchronous systems where the functional correctness of a circuit also depends on timing considerations. For example, for a synchronous system one has to calculate the worst-case delay for each part of the system and for any computation in order to satisfy the requirement that this delay must be smaller than the clock period.

As a last reason, we believe that the translation of parallel programs into delay-insensitive circuits offers a number of advantages compared to the translation of parallel programs into synchronous systems. In [5] a method is presented with which the synchronization and communication between parallel parts of a system can be programmed and realized in a natural way.

4. Directed trace structures and commands

In this and the next sections we describe the underlying formalism for the design of delay-insensitive circuits. Components, i.e. abstractions of circuits, are specified by so-called *directed trace structures* satisfying certain properties. Before we give a number of such specifications for components, we briefly explain what directed trace structures are and how they can be constructed similarly to regular expressions.

4.0. Directed trace structures

A *directed trace structure* is a pair $<A,B,X>$, where A and B are finite sets of symbols and $X \subseteq (A \cup B)^*$. The set $(A \cup B)^*$ is the set of all finite-length sequences of symbols from $A \cup B$. A finite sequence of symbols is called a *trace*. The empty trace is denoted by ϵ. Notice that $\varnothing^* = \{\epsilon\}$. For a directed trace structure $R = <A,B,X>$, the set $A \cup B$ is called the *alphabet* of R and denoted by $\mathbf{a}R$; the set A is called the *input alphabet* of R and denoted by $\mathbf{i}R$; the set B is called the *output alphabet* of R and denoted by $\mathbf{o}R$; the set X is called the *trace set* of R and denoted by $\mathbf{t}R$.

Notational convention. In the following, directed trace structures are denoted by the capitals R, S, and T; traces are denoted by the lower-case letters r, s, and t; alphabets are denoted by the capitals A and B; symbols are usually denoted by the lower-case letters with exception of r, s, and t.
□

Remark In addition to directed trace structures, we also have (undirected) trace structures which are defined as pairs $<A, X>$, where $X \subseteq A^*$. The sets A and X are called the alphabet and the trace set of the trace structure respectively. In this paper we consider directed trace structures only .
□

4.1. Operations on directed trace structures

The definitions and notations for the operations *concatenation, union, repetition, (taking the) prefix-closure, projection,* and *weaving* of directed trace structures are as follows.

$$R;S = <\mathbf{i}R \cup \mathbf{i}S\ ,\ \mathbf{o}R \cup \mathbf{o}S\ ,\ \mathbf{t}R\,\mathbf{t}S>$$
$$R|S = <\mathbf{i}R \cup \mathbf{i}S\ ,\ \mathbf{o}R \cup \mathbf{o}S\ ,\ \mathbf{t}R \cup \mathbf{t}S>$$
$$[R] = <\mathbf{i}R\ ,\ \mathbf{o}R\ ,\ (\mathbf{t}R)^*>$$
$$\mathbf{pref}\,R = <\mathbf{i}R\ ,\ \mathbf{o}R\ ,\ \{s\,|\,(\mathbf{E}t::st \in \mathbf{t}R)\}>$$
$$R{\restriction}A = <\mathbf{i}R \cap A\ ,\ \mathbf{o}R \cap A\ ,\ \{t{\restriction}A\,|\,t \in \mathbf{t}R\}>$$
$$R\|S = <\mathbf{i}R \cup \mathbf{i}S\ ,\ \mathbf{o}R \cup \mathbf{o}S\ ,\ \{t \in (\mathbf{a}R \cup \mathbf{a}S)^*\,|\,t{\restriction}\mathbf{a}R \in \mathbf{t}R \wedge t{\restriction}\mathbf{a}S \in \mathbf{t}S\}>,$$

where $t{\restriction}C$ denotes the trace t projected on C, i.e. the trace t from which all symbols not in C have been deleted. Concatenation of sets is denoted by juxtaposition and $(\mathbf{t}R)^*$ denotes the set of all finite-length concatenations of traces in $\mathbf{t}R$.

The operations concatenation, union, and repetition are familiar operations from formal language theory. We have added three operations: prefix-closure, projection, and weaving.

The **pref** operator constructs prefix-closed trace structures. A trace structure R is called *prefix-closed* if $\mathbf{pref}\,R = R$ holds. Later, we use prefix-closed and non-empty directed trace structures for the specification of components. A non-empty trace structure is a trace structure R for which $\mathbf{t}R \neq \varnothing$

The projection operator allows us to abstract away from a set of 'internal' symbols.

The weave operation constructs trace structures whose traces are weaves of traces from the constituent trace structures. Notice that common symbols must match, and, accordingly, weaving expresses 'instantaneous' synchronization. The set of symbols on which this synchronization takes place is the intersection of the alphabets.

The weave of n trace structures $R.i$, $0 \leqslant i < n$, is denoted by $(\|i:0 \leqslant i < n : R.i)$. A similar notation holds for the union of alphabets $A.i$, $0 \leqslant i < n$, which is denoted by $(\cup i:0 \leqslant i < n : A.i)$.

4.2. *Directed commands*

A directed trace structure is called a *regular directed trace structure* if its trace set is a regular set, i.e. a set generated by some regular expression. A *directed command* is a notation similar to regular expressions for representing a regular directed trace structure.

Let U be a sufficiently large set of symbols. The characters ϵ, $\varnothing$, $b?$, $b!$, and $!b?$ with $b \in U$, are called *atomic directed commands*. They represent the atomic directed trace structures $<\varnothing,\varnothing,\{\epsilon\}>$, $<\varnothing,\varnothing,\varnothing>$, $<\{b\},\varnothing,\{b\}>$, $<\varnothing,\{b\},\{b\}>$, $<\{b\},\{b\},\{b\}>$, respectively. Every atomic directed command and every expression for a directed trace structure constructed from the atomic directed commands and finitely many applications of the operations defined in Section 4.1 is called a *directed command*. In such an expression parentheses are allowed. For example, the expression $(a?\|b?);c!$ is a directed command and represents the directed trace structure $<\{a,b\},\{c\},\{abc,bac\}>$.

NOTATIONAL CONVENTION. In the following, directed commands are denoted by capital E's. The input and output alphabet and the trace set of the directed trace structure represented by command E are denoted by $\mathbf{i}E$, $\mathbf{o}E$, and $\mathbf{t}E$ respectively. In order to save on parentheses,

we stipulate the following priority rules for the operations just defined. Unary operators have highest priority. Of the binary operators in Section 4.1, weaving has highest priority, then concatenation, then union, and finally projection.
□

PROPERTY 4.2.0. *Every directed command represents a regular directed trace structure.*
□

EXAMPLE 4.2.1. Syntactically different commands can express the same trace structure. We have, for example,

$$\mathbf{pref}[a?;c!] \parallel \mathbf{pref}[b?;c!] = \mathbf{pref}[a?\|b?;c!]$$

$$\mathbf{pref}[a?;b!] \parallel \mathbf{pref}[a?;c!] = \mathbf{pref}[a?;b!\|c!]).$$

□

5. SPECIFYING COMPONENTS

This section addresses the specification of components, which may be viewed as abstractions of circuits. Components are specified by *directed trace structures* satisfying certain properties. In this paper we shall keep to regular components, i.e. to regular directed trace structures. We explain how a directed trace structure prescribes all possible communication behaviors between a component and *environment* at their mutual *boundary.* A number of basic components are then specified by means of directed commands.

5.0. Specifications and their interpretation

A communication behavior between component and environment is specified by a prefix-closed, non-empty, directed trace structure R with $\mathbf{i}R \cap \mathbf{o}R = \varnothing$. The alphabet of R contains the names of all terminals at which component and environment communicate with each other. This set of terminals is also called the *boundary* between component and environment. An occurrence of a communication action at a terminal is represented by the name of that terminal. We assume that an occurrence of a communication action is determined by only one of the communicants. The sets $\mathbf{i}R$ and $\mathbf{o}R$ are used to stipulate by whom a communication action may be produced. The set $\mathbf{i}R$ contains all communication actions that may be produced by the environment and the set $\mathbf{o}R$ contains all communication actions that may be produced by the component. The trace set of R contains all communication behaviors that may take place between component and environment.

A communication behavior evolves by the production of communication actions. Because $\mathbf{i}R \cap \mathbf{o}R = \varnothing$, a communication action is produced either by the component or by the environment. Together, the sets $\mathbf{i}R, \mathbf{o}R$, and $\mathbf{t}R$ specify when which communication action may be produced and by whom as follows. Let the communication actions that have already taken place correspond to the trace $t \in \mathbf{t}R$, and let $tb \in \mathbf{t}R$. (Initially, $t = \epsilon$.) If $b \in \mathbf{i}R$, then the environment may produce a next communication action b; if $b \in \mathbf{o}R$, then the

component may produce a next communication action b. These are also the only rules for the production of inputs and outputs for environment and component respectively.

EXAMPLE 5.0.0 Consider the command E given by $E = \mathbf{pref}[a?||b?;c!]$. This command specifies a component for which the environment initially produces the inputs a and b in parallel (or in arbitrary order). Then the component may produce output c. Only after output c has been produced may the environment produce inputs a and b again. The component may then produce output c again and this behavior repeats.
□

Because the directed trace structure R specifies the communication behavior of both component and environment, we speak of component R and environment R. The role of component and environment can be interchanged by *reflecting* R:

DEFINITION 5.0.1. The *reflection* of R, denoted by $\overline{R}$, is defined by

$$\overline{R} = \langle \mathbf{o}R,\ \mathbf{i}R,\ \mathbf{t}R \rangle .$$

□

(Consequently, $\mathbf{i}\overline{R} = \mathbf{o}R$, $\mathbf{o}\overline{R} = \mathbf{i}R$, and $\mathbf{t}\overline{R} = \mathbf{t}R$.) Instead of environment R we can now also speak of component $\overline{R}$.

EXAMPLE 5.0.2. The environment E, where $E = \mathbf{pref}[a?||b?;c!]$, can be expressed as component $\overline{E}$, where $\overline{E} = \mathbf{pref}[a!||b!;c?]$.
□

With the above interpretation of a specification we explicitly prescribe restrictions on how the environment may communicate with the component. Later, in Section 6, when we are interested in realizing components by connections of other components, we assume that the environment of this connection behaves as prescribed. Under this assumption the connection has to react as prescribed for the component. In case of a physical implementation of a component (e.g. for basic components) the environment stipulates under what conditions correct physical operation must be guaranteed.

A possible physical implementation of a component is that each symbol b of $\mathbf{a}R$ corresponds to a terminal of a circuit, and each occurrence of b in a trace of $\mathbf{t}R$ corresponds to a voltage transition at that terminal. There is no distinction between high-going and low-going transitions: both transitions are denoted by the same symbol. Outputs are transitions caused by the circuit and inputs are transitions caused by the environment. When we refer to implementations in this paper we mean the above mentioned implementations and we assume that initially the voltage levels at the terminals are low.

In the following subsections, a number of components are specified by directed commands. For each of these components we also give a pictorial representation, called a *schematic*.

5.1. Specification of a set of basic components

In Figure 5.0 a set of basic components is specified by means of directed commands. The first column lists the names of the components, the second column the specifications, and the third column the schematics.

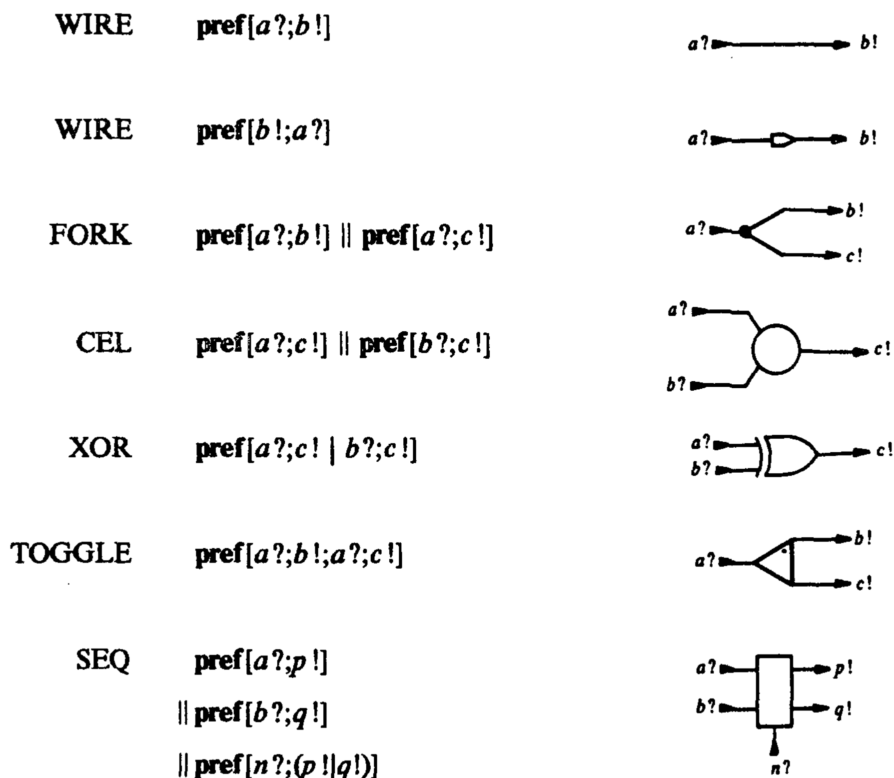

FIGURE 5.0. Specification of a set of basic components.

There are two WIRE components. A WIRE component describes the transmission of a signal from terminal to terminal. Notice that both WIRE components have the same behavior except for a difference in initial states. For the WIRE component **pref**[a?;b!] the environment initially produces an input a. For the WIRE component **pref**[b!;a?] the component initially produces an output b. This difference in initial states (or the production of initial symbols) is depicted by an open arrow head in a schematic. The trace structures are, apart from a renaming, each other's reflection.

Operationally speaking, the first WIRE component corresponds to a physical wire. Notice that there is always at most one transition propagating along either WIRE component according to our interpretation of a specification as a prescription for both component and environment.

A FORK component performs the primitive operation of duplication of inputs.

A CEL component performs the primitive operation of synchronization on an output. (Recall from example 4.2.1 that a CEL and FORK component can be represented by other directed commands.) CEL components are reflections of FORK components, apart from a different initialization. (The CEL component can be implemented by the Muller *C*-element, named after David E. Muller.)

The TOGGLE component determines the parity of the input occurrences. After each odd occurrence of input a output b is produced and after each even occurrence of input a output c is produced.

The XOR component 'merges' two inputs into one output. Notice that for this component the environment produces either an input a or an input b. In both cases the component will then produce an output c, after which the environment may produce a next input again. (The XOR component can be implemented by an exclusive OR gate.)

The 2-SEQ component is a kind of arbiter component. For a 2-SEQ component we use the following terminology. Output p is called the *grant* of *request* a. Similarly, output q is the grant of request b. We say that request a is *pending* after trace t if $t\mathrm{N}a - t\mathrm{N}p = 1$, where $t\mathrm{N}x$ denotes the number of x's in trace t. A SEQ component grants one request for each occurrence of input n. We also say that the SEQ component *sequences* the grants and, for this reason, it is sometimes also called a *sequencer*. In sequencing the grants it may have to arbitrate among several pending requests. If there is only one pending request after receipt of input n, no arbitration is needed and the grant for this pending request will be produced. If, however, there are two pending requests after receipt of input n, the SEQ component has to arbitrate which of the pending requests will be granted.

5.2. Examples

EXAMPLE 5.2.0. Consider the modulo-3 counter specified by the following communication behavior. The modulo-3 counter has three communication actions: one input, denoted by a, and two outputs, denoted by p and q. The communication behavior is an alternation of inputs and outputs, starting with an input. The outputs depend on the inputs as follows. After the n-th input, where $n > 0$ and $n \,\mathbf{mod}\, 3 \neq 0$, output q is produced; if $n \,\mathbf{mod}\, 3 = 0$, then output p is produced. This behavior is expressed in the following directed command $E0$, where

$$E0 \;=\; \mathbf{pref}[a?;q!;a?;q!;a?;p!].$$

Notice that the TOGGLE component of Figure 5.0 can be considered as a modulo-2 counter. In Section 6 we discuss a decomposition of the modulo-3 counter into basic components of Figure 5.0. (Before reading this section, the reader may try to find such a connection.)

□

EXAMPLE 5.2.1. This example concerns the specification of the communication behavior of a token-ring interface. Consider a number of machines. For each machine we introduce a component, and all components are connected in a ring. Through this ring a so-called token is propagated from component to component. The ring-wise connection is called a *token*

ring, and the components are called *token-ring interfaces*. Each machine communicates with the token ring through its token-ring interface.

In order to achieve mutual exclusion among machines entering a critical section, the following protocol is described for a token-ring interface. The schematic of the token-ring interface is given in Figure 3.

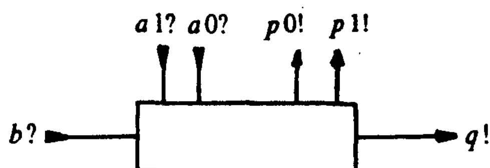

FIGURE 3. A token-ring interface.

The communication actions between token-ring interface and machine are interpreted as follows.

$a1?$ request for the token by the machine

$p1!$ grant of the token to the machine

$a0?$ release of the token by the machine

$p0!$ confirm of release.

With respect to these actions the protocol satisfies the specification **pref**$[a1?;p1!;a0?;p0!]$.

The communication actions between token-ring interface and the rest of the token ring are interpreted as follows.

$b?$ receipt of the token

$q!$ sending of the token.

With respect to these actions the protocol satisfies the specification **pref**$[b?;q!]$.

The synchronization between the two protocols must satisfy the following requirements. After each receipt of the token, the token can either be sent on to the next token-ring interface or, if there is also a request from the machine, the token can be granted to the machine. If the machine releases the token, it is sent on to the next token-ring interface. The complete communication protocol for the token-ring interface can be specified by the directed command

pref$[a1?;p1!;a0?;p0!]$

$\|$ **pref**$[b?;(q! \mid p1!;a0?;q!)]$.

Recall that weaving denotes synchronization on common symbols and therefore alternative $p1!;a0?;q!$ of the second line can only be executed when in the first line $p1!$ can be executed as well. In the following sections we show how this synchronization can be realized by a delay-insensitive connection of basic elements. (As an exercise, the reader may try to

find such a connection before reading Section 6.)
□

6. Decomposition

The idea of this paper is to realize a component by means of a delay-insensitive connection of basic components. In this section we formalize this idea by presenting the definitions and theorems underlying this approach.

First, we define what we mean by 'a component can be realized by a connection of (other) components such that the functional operation of the connection is insensitive to delays in the components'. This is formulated in the definition of *decomposition.*

Subsequently, we give two theorems on decomposition: the *Substitution Theorem,* which enables us to decompose a component in a hierarchical way, and the *Separation Theorem,* which enables us to decompose parts of a specification separately.

In a decomposition delays may occur in a component but the communications between components are considered to be instantaneous and unidirectional. Usually these communications take place through wires in which delays can also be incurred. If we incorporate wire delays as well in a decomposition then we obtain the definition of *DI decomposition.* A DI decomposition corresponds to a realization of a component by means of a connection of components such that the functional operation of the connection is insensitive to delays in components and in communications (i.e. wires).

In order to relate decomposition and DI decomposition we introduce *DI components.* Decomposition and DI decomposition are in general not equivalent, but if all constituting components are so-called DI components, then decomposition and DI decomposition are equivalent. A DI component may be interpreted as a component whose communication behavior with its environment is insensitive to wire delays.

6.0. The definition

Below, we first present the definition of decomposition and then give a brief motivation for it.

DEFINITION 6.0.0. *Let* $n>1$. *We say that component* $S.0$ *can be decomposed into the components* $S.i$, $1\leqslant i<n$, *denoted by*

$$S.0 \rightarrow (i: 1\leqslant i<n: S.i),$$

if the following conditions are satisfied.
Let $R.0=\overline{S.0}$, $R.i=S.i$ *for* $1\leqslant i<n$, *and* $W=(\|i: 0\leqslant i<n: R.i)$.

(*i*) *(Closed connection)*
$(\cup i: 0\leqslant i<n: \mathbf{o}(R.i)) = (\cup i: 0\leqslant i<n: \mathbf{i}(R.i))$.

(*ii*) *(No output interference)*
$\mathbf{o}(R.i)\cap\mathbf{o}(R.j)=\varnothing$ *for* $0\leqslant i,j<n \wedge i\neq j$.

(*iii*) (*Connection behaves as specified at boundary* $\mathbf{a}(S.0)$)
$\mathbf{t}W{\upharpoonright}\mathbf{a}(R.0) = \mathbf{t}(R.0)$.

(*iv*) (*Connection is free of computation interference*)
For all traces t, symbols x, and indexes i, $0 \leq i < n$, *we have*
$t \in \mathbf{t}W \wedge x \in \mathbf{o}(R.i) \wedge tx{\upharpoonright}\mathbf{a}(R.i) \in \mathbf{t}(R.i) \;\Rightarrow\; tx \in \mathbf{t}W$.

□

NOTATIONAL REMARK. The notation $(i : 0 \leq i < n : S.i)$ can be interpreted as an enumeration of the components $S.i$, $0 \leq i < n$. Notice, however, that the order of this enumeration is not important, as can be deduced from the definition of decomposition. Instead of, for example, $S.0 \rightarrow (i : 1 \leq i < 4 : S.i)$ we sometimes write $S.0 \rightarrow S.1, S.2, S.3$. Here, the comma separates the components.
□

In Section 5, we stipulated that a directed trace structure $S.0$ prescribes the joint behavior of component and environment: it specifies when the component may produce outputs and when the environment may produce inputs. In a decomposition of component $S.0$ we require that the production of outputs of component $S.0$ is realized by a connection of components. We assume that the environment of this connection produces the inputs as specified for environment $S.0$. This environment can also be seen as component $\overline{S.0}$. Accordingly, in order to comprise all components that produce outputs relevant to the decomposition, we consider the connection of components $\overline{S.0}$ and $S.i$, $1 \leq i < n$.

Condition (*i*) says that there are no dangling inputs and outputs in the connection: every output is connected to an input, and every input is connected to an output. We call such a connection a *closed connection*.

Condition (*ii*) requires that outputs of distinct components are not connected with each other. If (*ii*) holds we say that the connection is *free of output interference*.

Condition (*iii*) requires that the behavior of the connection at the boundary $\mathbf{a}(S.0)$ behaves as specified by $\mathbf{t}(S.0)$. The behavior of the connection is given by $\mathbf{t}W = \mathbf{t}(\|i : 0 \leq i < n : R.i)$. Restriction of this behavior to the boundary $\mathbf{a}(S.0)$ $(= \mathbf{a}(R.0))$ is expressed by $\mathbf{t}W{\upharpoonright}\mathbf{a}(R.0)$.

Condition (*iv*) requires that the connection is free of computation interference. We say that the connection has danger of *computation interference*, if there exists a trace t, symbol x, and index i, $0 \leq i < n$, such that

$$t \in \mathbf{t}W \wedge x \in \mathbf{o}(R.i) \wedge tx{\upharpoonright}\mathbf{a}(R.i) \in \mathbf{t}(R.i) \wedge tx \notin \mathbf{t}W.$$

In words, if after a mutually agreed behavior a component can produce an output that is not in accordance with the prescribed behavior of other components, then we say that the connection has danger of computation interference. Absence of computation interference guarantees that every output that can be produced can also be received as input, i.e. no environment prescription is violated.

A violation of an environment prescription of a component can cause hazardous behavior.

For example, if WIRE component $\mathbf{pref}[a?;b!]$ receives two inputs a without producing an output b, we have computation interference for the WIRE component (caused by the environment). Operationally speaking, in the case of this computation interference more than one transition is propagating along a wire, which can cause hazardous behavior and must, therefore, be avoided.

REMARK 0. Some misbehaviors of circuits that are characterized in classical switching theory by hazards or critical races can be seen as special cases of computation interference. Absence of interference in a decomposition guarantees that the thus synthesized circuit is free of hazards and critical races, provided that the basic components used are themselves free of hazards and critical races.
□

REMARK 1. In the definition of decomposition we have not included conditions such as absence of deadlock and livelock. This is done for reasons of simplicity but also because we believe that these issues can be dealt with separately. For a study on these issues we refer to [9].
□

Notice that we have described decomposition as a goal-directed activity: we start with a component $S.0$ and try to find components $S.i$, $1 \leqslant i < n$, such that the relation $S.0 \rightarrow (i: 1 \leqslant i < n: S.i)$ holds. Thus, we explicitly use the assumption that the environment of the connection of components behaves as specified for environment $S.0$. We do not start with components $S.i$, $1 \leqslant i < n$, to find out what could be made of them without requiring anything from the environment. This is also the reason why this method is called decomposition instead of composition. Also, a decomposition does not have to be unique. For a component several decompositions may exist.

EXAMPLE 6.0.1. We demonstrate that the modulo-3 counter of Example 5.2.0 can be decomposed into the two TOGGLE components and an XOR component. A schematic of this decomposition is given in Figure 6.0.

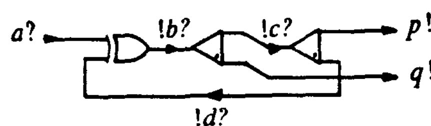

FIGURE 6.0. A decomposition of the modulo-3 counter.

To verify this decomposition we take

$$R.0 = \mathbf{pref}[a!;q?;a!;q?;a!;p?],$$

$$R.1 = \mathbf{pref}[(a? \mid d?);b!],$$

$$R.2 = \mathbf{pref}[b?;q!;b?;c!], \text{and}$$

$$R.3 = \mathbf{pref}[c?;d!;c?;p!],$$

By inspection, we observe that the connection of the components $R.0$, $R.1$, $R.2$, and $R.3$ is closed and free of output interference.

Furthermore, we have

$$\begin{aligned} \mathbf{t}W &= \mathbf{t}(R.0 \,||\, R.1 \,||\, R.2 \,||\, R.3) \\ &= \mathbf{tpref}[!a?;!b?;!q?;!a?;!b?;!c?;!d?;!b?;!q?;!a?;!b?;!c?;!p?]. \end{aligned}$$

From this we derive that $\mathbf{t}W\!\restriction\!\mathbf{a}(R.0) = \mathbf{t}(R.0)$. Accordingly we conclude that the connection behaves as specified at the boundary $\mathbf{a}(R.0)$.

For absence of computation interference we have to prove for all t, x, i, $0 \leq i < 4$, that

$$t \in \mathbf{t}W \wedge x \in \mathbf{o}(R.i) \wedge tx\!\restriction\!\mathbf{a}(R.i) \in \mathbf{t}(R.i) \;\Rightarrow\; tx \in \mathbf{t}W.$$

Instead of proving this for all triples (t,x,i), we take for all states of $\mathbf{t}W$ a representative t and consider all x and i, $0 \leq i < 3$, such that

$$t \in \mathbf{t}W \wedge x \in \mathbf{o}(R.i) \wedge tx\!\restriction\!\mathbf{a}(R.i) \in \mathbf{t}(R.i). \tag{6.0}$$

(For the states of $\mathbf{t}W$ we may take the equivalence classes of the right-invariant relation under concatenation, but also the states of any finite automaton accepting $\mathbf{t}W$.) It suffices to prove for these triples (t, x, i) that $tx \in \mathbf{t}W$. By inspection, we find that for the triples satisfying (6.0) indeed $tx \in \mathbf{t}W$. Consequently, we conclude that Figure 6.0 gives a decomposition of the modulo-3 counter.

□

EXAMPLE 6.0.2. Similar to Example 6.0.1 we can prove the decompositions

$$\mathbf{pref}[a1?;p1!;a0?;a1!] \;\rightarrow\; \mathbf{pref}[a1?;p1!]\,,\; \mathbf{pref}[a0?;p0!] \tag{6.1}$$

$$\mathbf{pref}[b?;(q1! \mid p1!;a0?;q0!] \;\rightarrow\; \mathbf{pref}[b?;(q1! \mid p1!)]\,,\; \mathbf{pref}[a0?;q0!]. \tag{6.2}$$

and

$$\begin{aligned} &\mathbf{pref}[b?;(q! \mid p1!;a0?;q!)] \\ \rightarrow\; &\mathbf{pref}[b?;(q1! \mid p1!;a0?;q0!]\,,\; \mathbf{pref}[(q1? \mid q0?);q!]\,. \end{aligned} \tag{6.3}$$

In decomposition (6.3) we have made a distinction between the outputs q in the two alternatives by renaming them into $q1$ and $q0$. By means of a XOR component these two symbols can then be merged again into output q.

□

EXAMPLE 6.0.3. In this example we consider a decomposition of the component specified by

$$E0 = \mathbf{pref}[a?;b!;c?\|d?;e!;a?\|c?;b!;d?;e!].$$

This component can be decomposed into the components

$$E1 = \textbf{pref}[a?;b!;c?;a?\|c?;b!] \text{ and}$$

$$E2 = \textbf{pref}[c?\|d?;e!].$$

Component $E2$ is a CEL component and component $E1$ can be implemented by an OR gate, assuming that initially all voltage levels are zero. The decomposition of $E0$ is depicted in Figure 6.1, where we have taken the schematic of the OR gate as the schematic for component $E1$.

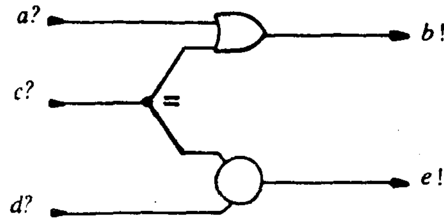

FIGURE 6.1. A decomposition of $E0$.

In Figure 6.1 we have used a fork with an equality sign next to the fat dot. The equality sign signifies that this fork is not a genuine FORK component of the decomposition, but that both components to which this fork is connected have the same input, viz. input c (More will be said about these forks in the next example.)
□

EXAMPLE 6.0.4. Suppose that we would consider the fork in Figure 6.1 of the previous example as a genuine FORK component of a tentative decomposition of $E0$. The schematic of this tentative decomposition is given in Figure 6.2

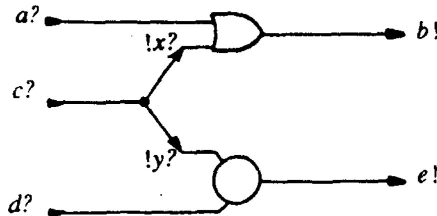

FIGURE 6.2. A tentative decomposition of $E0$.

We demonstrate that this tentative decomposition is not a decomposition: it has danger of computation interference. This is shown as follows. Let

$$R.0 = \textbf{pref}[a!;b?;c!\|d!;e?;a!\|c!;b?;d!;e?]$$

$$R.1 = \textbf{pref}[a?;b!;x?;a?\|x?;b!]$$

$R.2 = \mathbf{pref}[d?||y?;e!]$

$R.3 = \mathbf{pref}[c?;x!||y!]$.

Let, furthermore, $W = R.0||R.1||R.2||R.3$. For trace $t = abcdye$ we observe $t \in \mathbf{t}W$. Moreover, we have that after trace t component $R.0$ can produce output a, i.e. $a \in \mathbf{o}(R.0) \wedge ta \upharpoonright \mathbf{a}(R.0) \in \mathbf{t}(R.0)$. But from the specification of $R.1$ we observe that input a can not be received after trace t, i.e. $aba \notin \mathbf{t}(R.1)$. Consequently, $ta \notin \mathbf{t}W$, and we conclude that the connection has computation interference. (Notice that, indeed, if we may assume arbitrary wire delays hazardous behavior can occur at output b. Notice also that the other three conditions of decomposition are satisfied for the components $R.0$, $R.1$, $R.2$, and $R.3$.)

In order to avoid hazardous behavior in an implementation of the decomposition of $E0$ into $E1$ and $E2$, as given in the previous example, the fork in Figure 6.1 must be implemented by a so-called *isochronic fork* [12, 13]. An isochronic fork is a (physical) fork in which the delays in the branches of the fork are (much) smaller than the delays in the elements to which the fork is connected.

□

6.1. Some theorems on decomposition

As the reader may have noticed in the above examples, verifying whether a connection of components forms a decomposition of a component is simple but may be laborious. It would be convenient if we would have some theorems on decomposition with which the verification, or even the derivation, of a decomposition could be done quickly.

A theorem helpful in finding decompositions of a component is the Substitution Theorem. This theorem applies to problems of the following kind. Suppose that component $S.0$ can be decomposed into a number of components of which T is one such component. Suppose, moreover, that T can be decomposed further into a number of components. Under what conditions can the decomposition of T be substituted in the decomposition of $S.0$? We have

THEOREM 6.1.0. (Substitution Theorem)
Let components $S.0$, $S.1$, $S.2$, $S.3$, and T satisfy

$$(\mathbf{a}(S.0) \cup \mathbf{a}(S.1)) \cap (\mathbf{a}(S.2) \cup \mathbf{a}(S.3)) = \mathbf{a}T. \tag{6.4}$$

We have

$$\begin{aligned} & S.0 \to S.1, T \\ & \wedge T \to S.2, S.3 \\ \Rightarrow\ & S.0 \to S.1, S.2, S.3. \end{aligned}$$

□

The proof of this theorem can be found in [5] (pp. 47).

Condition (6.4) of the above theorem states that the only symbols that the two

decompositions have in common are symbols from **a**T. Condition (6.4) is essentially a void condition, since, by an appropriate renaming of the internal symbols in the decomposition of T, this condition can always be satisfied. The internal symbols of the decomposition of T are given by $(\mathbf{a}(S.2) \cup \mathbf{a}(S.3)) \setminus \mathbf{a}T$.

The above theorem considers decompositions into two components only. The generalization of this theorem to decompositions into more than two components is straightforward and omitted here.

NOTATIONAL REMARK. In the derivation of a decomposition of a component in a number of steps we use the following notation.

$$
\begin{aligned}
& S.0 \\
\rightarrow & \{\text{hint why } S.0 \rightarrow S.1, T\} \\
& S.1, T \\
\rightarrow & \{\text{hint why } T \rightarrow S.2, S.3\} \\
& S.1, S.2, S.3.
\end{aligned}
$$

Such a derivation is then based on the Substitution Theorem, and care must be taken that the condition for its application holds.
□

EXAMPLE 6.1.1. Applying the Substitution Theorem to Example 6.0.2, we derive

$$
\begin{aligned}
& \mathbf{pref}[b?;(q! \mid p1!;a0?;q!)] \\
\rightarrow & \{(6.3)\} \\
& \mathbf{pref}[b?;(q1! \mid p1!;a0?;q0!)]\,,\ \mathbf{pref}[(q1? \mid q0?);q!] \\
\rightarrow & \{(6.2)\} \\
& \mathbf{pref}[b?;(q1! \mid p1!)]\,,\ \mathbf{pref}[a0?;q0!]\,,\ \mathbf{pref}[(q1? \mid q0?);q!].
\end{aligned}
$$

□

Another theorem that may be convenient in finding a decomposition of a component is the Separation Theorem. We have

THEOREM 6.1.2 (Separation Theorem) *For components $S.i$ and $T.i$, $0 \leq i < n$, we have*

$$
\begin{aligned}
& S.0 \rightarrow (i: 1 \leq i < n: S.i) \\
\wedge\ & T.0 \rightarrow (i: 1 \leq i < n: T.i) \\
\Rightarrow\ & S.0 \| T.0 \rightarrow (i: 1 \leq i < n: S.i \| T.i)
\end{aligned}
$$

if the following conditions are satisfied.

$$A \cap B = \varnothing \tag{6.5}$$

$$Out.i \cap Out.j = \varnothing \quad \text{for } 0 \leqslant i,j < n \wedge i \neq j, \tag{6.6}$$

where

$A = (\cup i: 1 \leqslant i < n: \mathbf{a}(S.i)) \setminus \mathbf{a}(S.0)$
$B = (\cup i: 1 \leqslant i < n: \mathbf{a}(T.i)) \setminus \mathbf{a}(T.0)$
$Out.i = \mathbf{o}(\underline{S.i}) \cup \mathbf{o}(\underline{T.i})$ *for* $1 \leqslant i < n$, *and*
$Out.0 = \mathbf{o}(\overline{S.0}) \cup \mathbf{o}(\overline{T.0})$.

□

The proof of this theorem can be found in [5] (cf pp. 51).

Condition (6.5) can be interpreted as 'the internal symbols of the decompositions are row-wise disjoint', where the internal symbols of the decomposition of $S.0$ (i.e. row 0) are given by A. This condition can always be satisfied by an appropriate renaming of the internal symbols.

Condition (6.6) can be interpreted as 'the outputs are column-wise disjoint', where the outputs of column $\underline{i, 0 \leqslant i < n,}$ are given by $Out.i$. (Notice that $Out.0$ represents the outputs of the components $\overline{S.0}$ and $\overline{T.0}$.) Recall from the definition of decomposition that the ordering of the components in $(i: 1 \leqslant i < n: S.i)$ is not relevant. Accordingly, we may reorder the components such that condition (6.6) can be satisfied.

Theorem 6.1.2 can be generalized in a natural way to decompositions of weaves of more than two directed trace structures.

EXAMPLE 6.1.3. We apply the Separation Theorem to obtain a decomposition of the token-ring interface. The token-ring interface was specified by

$$\mathbf{pref}[a1?;p1!;a0?;p0!]$$
$$\| \mathbf{pref}[b?;(q! \mid p1!;a0?;q!)].$$

This command is written a a weave of two commands $E0$ and $E1$, where

$$E0 = \mathbf{pref}[a1?;p1!;a0?;p0!] \text{ and}$$
$$E1 = \mathbf{pref}[b?;(q! \mid p1!;a0?;q!)].$$

From Examples 6.0.2 and 6.1.1 we derive that $E0$ and $E1$ can be decomposed as follows.

$$E0 \rightarrow \mathbf{pref}[a1?;p1!], \mathbf{pref}[a0?;p0!], \epsilon, \epsilon$$
$$E1 \rightarrow \mathbf{pref}[b?;(q1! \mid p1!)], \epsilon, \mathbf{pref}[a0?;q0!], \mathbf{pref}[(q1? \mid q0?);q!].$$

We have added several (epsilon) components ϵ in order to bring the decompositions into a form to which the Separation theorem can be applied. Adding (epsilon) components does not invalidate a decomposition.

Verifying conditions (6.3) and (6.4) of the Separation Theorem, we derive that the internal symbols of the decompositions are disjoint, since $\varnothing \cap \{q0, q1\} = \varnothing$. Furthermore, we observe that the outputs are column-wise disjoint. Consequently, we conclude by the

Separation Theorem that

$E0 \| E1$

$\rightarrow$ {Above decompositions, Separation Theorem}

$\textbf{pref}[a1?;p1!] \parallel \textbf{pref}[b?;(q1! \mid p1!)]$

$,\ \textbf{pref}[a0?;p0!]$

$,\ \textbf{pref}[a0?;q0!]$

$,\ \textbf{pref}[(q1? \mid q0?);q!].$

All components in this list are basic components except for the one in the first line. This command almost looks like the command of a SEQ component. We are missing a command expressing the alternation of a request (for grant $q1$) and grant $q1$. Therefore, we introduce the symbol $rq1$, representing the request for grant $q1$, and apply the following decomposition.

$\textbf{pref}[a1?;p1!] \parallel \textbf{pref}[b?;(q1! \mid p1!)]$

$\rightarrow$ {Def. of decomposition, introduction of internal symbol $rq1$}

$\textbf{pref}[a1?;p1!] \parallel \textbf{pref}[rq1?;q1!] \parallel \textbf{pref}[b?;(q1! \mid p1!)]$

$,\ \textbf{pref}[rq1!;q1?].$

Applying the Substitution Theorem once more we obtain the complete decomposition of the token-ring interface into basic elements:

$E0 \| E1$

$\rightarrow$ {Decomposition above, Substitution Theorem}

$\textbf{pref}[a1?;p1!] \parallel \textbf{pref}[rq1?;q1!] \parallel \textbf{pref}[b?;(q1! \mid p1!)]$

$,\ \textbf{pref}[rq1!;q1?]$

$,\ \textbf{pref}[a0?;p0!]$

$,\ \textbf{pref}[a0?;q0!]$

$,\ \textbf{pref}[(q1? \mid q0?);q!].$

This decomposition is shown in Figure 6.3. Since both $a0$ and $q1$ are inputs for two components, we have drawn Figure 6.3 two forks to depict the decomposition. Notice that these forks do not occur as genuine FORK components in the decomposition. For this reason, these forks should be drawn as isochronic forks. Later, in example 6.3.2, we shall see, however, that these forks can be considered as genuine FORK components in the decomposition. That is, operationally speaking, arbitrary delays may take place in the branches of the forks without affecting the functional behavior of the connection.

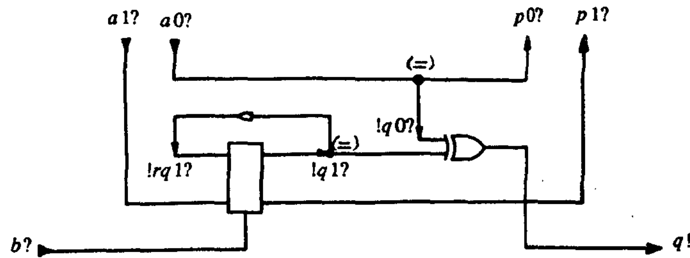

FIGURE 6.3 A decomposition of the token-ring interface.

□

6.2. DI decomposition

In our operational interpretation, a decomposition is intended to represent a decomposition of a physical circuit into sub-circuits. In these sub-circuits arbitrary delays may occur between the receipt of an input and the production of an output (with a different name). It is assumed, however, that the communications between the sub-circuits are instantaneous, i.e. the sending and receipt of a signal coincide. If these sub-circuits are connected by wires with unspecified delays this assumption is clearly violated, unless we explicitly include the connection wires in the connection of the sub-circuits. For this reason, we introduce WIRE components in a DI decomposition. We make the wire connections through 'intermediate terminals' as depicted in Figure 6.4. The set of intermediate terminals is called the ***intermediate boundary***.

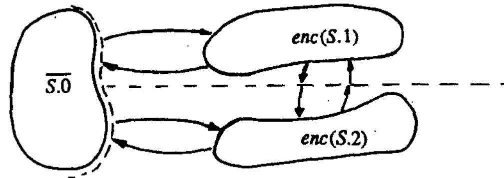

FIGURE 6.4. DI decomposition.

Operationally speaking, the WIRE components introduce delays in the communications between components and the intermediate boundary. Thus, they may affect the functional behavior of the connection (of components). If this closed connection operates as specified and is free of interference, then we call such a connection a delay-insensitive connection.

The formalization of a delay-insensitive connection of components is done as follows. Let $\mathbf{a}(S.k)$, $1 \leq k < n$, stand for an intermediate boundary and define the *enclosure* $enc(S.k)$ of this boundary by

$enc(S.k)$ is the trace structure obtained by replacing

each output a in $S.k$ by oa_k and

each input a in $S.k$ by ia_k.

Instead of considering the components $S.k$, $1 \leq k < n$, we consider the components $enc(S.k)$ and the intermediate boundaries given by $\mathbf{a}(S.k)$. For each k, $1 \leq k < n$ and $a \in \mathbf{a}(S.k)$ we introduce the WIRE component $Wire(k,a)$ between the boundary of the enclosure and the intermediate boundary by

$$\begin{aligned} Wire(k,a) &= \mathbf{pref}[oa_k?;a!] \quad \text{if } a \in \mathbf{o}(S.k) \\ &= \mathbf{pref}[a?;ia_k!] \quad \text{if } a \in \mathbf{i}(S.k). \end{aligned}$$

The collection of WIRE components for $S.k$, $1 \leq k < n$, is defined by

$$Wires(S.k) = (a : a \in \mathbf{a}(S.k) : Wire(k,a))$$

With these definitions we can formulate

DEFINITION 6.2.0. *We say that the components $S.k$, $1 \leq k < n$ form a DI decomposition of component $S.0$, denoted by*

$$S.0 \overset{DI}{\rightarrow} (k : 1 \leq k < n : S.k),$$

if and only if

$$S.0 \rightarrow (k : 1 \leq k < n : enc(S.k),\ Wires(S.k)).$$

□

REMARK. The definition of DI decomposition given here is slightly different from the one given in [5]. The definition in [5] requires that the (closed) connection is also insensitive to wire delays introduced between the intermediate boundary $\mathbf{a}(S.0)$ and the environment $S.0$ (i.e. component $\overline{S.0}$). The definition given here is easier to formulate and to use than the one in [5].
□

EXAMPLE 6.2.1. Of all the examples on decomposition in Section 6.0 we could verify whether these decompositions are also DI decompositions. In general, this may be rather laborious, because of the extra WIRE components. In the next section we present a theorem with which we can verify quickly whether a decomposition is also a DI decomposition. It turns out that the decompositions of the modulo-3 counter and the token-ring interface are indeed DI decompositions. The decomposition given in Example 6.0.3, however, is not a DI decomposition. Essentially, this is demonstrated in Example 6.0.4, where we showed that there was danger of computation interference in case extra WIRE components are introduced in the communications.
□

6.3. DI components

In this paper we are interested in DI decompositions of a component. In general, DI decompositions are more difficult to verify or derive than decompositions because of all the (connection) WIRE components. The two decompositions are equivalent, however, if all constituting components are so-called DI components. DI components are defined by

DEFINITION 6.3.0. Component S is called a DI component, if

$$S \rightarrow Wires(S), enc(S).$$

□

We have

THEOREM 6.3.1. *If all components $S.i$, $0 \leqslant i < n$, are DI components, then*

$$S.0 \rightarrow (i: 1 \leqslant i < n: S.i) \equiv S.0 \overset{DI}{\rightarrow} (i: 1 \leqslant i < n: S.i).$$

□

REMARK The proof from the left-hand side to the right-hand side follows immediately from the Substitution Theorem and the definitions of DI component and DI decomposition. The other way is more complicated. A complete proof can be found in [5].

□

Definition 6.3.0 formalizes that the set of allowed external communication behaviors, i.e. the communications between component and environment, is insensitive to wire delays. The characterization of a DI component S by the property $S \rightarrow Wires(S)$, $enc(S)$ can be considered as a formalization of the so-called *Foam Rubber Wrapper* (FRW) principle. Formally speaking, the FRW principle states that the specification of a component is invariant under the extension of WIRE components. Operationally speaking, the FRW metaphor expresses that the circuit specified by S is embedded in a 'Foam Rubber Wrapper' formed by the connection wires. The boundaries of the FRW are constituted by $\mathbf{a}enc(S)$ and $\mathbf{a}S$, as depicted in Figure 6.5.

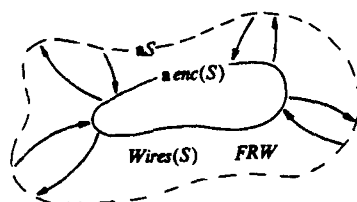

FIGURE 6.5. S is a DI component: $S \rightarrow Wires(S), enc(S)$.

The idea of formalizing delay-insensitivity by means of the FRW principle originates from Charles E. Molnar [17]. Jan Tijmen Udding was the first to give a rigorous formulation of this principle in terms of directed trace structures. In [26] he postulates a number of rules which a component must satisfy in order to meet the FRW principle. It turns out that Udding's definition of a DI component is equivalent to Definition 6.3.0 [5]. T.P. Fang had earlier expressed the FRW principle —though less completely— by means of Petri Net rules. In [20] another formalization of the FRW principle is given by Huub Schols. For a proof of the equivalence of Udding's and Schols's formalization we refer to [20, 21].

In order to use Theorem 6.3.1 we have to know whether a component is a DI component. The recognition of DI components can be done in several ways. We could use Definition 6.3.0, for example, or the rules given by Jan Tijmen Udding. Yet another method is to use a *DI grammar*, i.e. a grammar that generates commands representing DI components. Such a grammar is given in [5]. We shall not recapitulate these methods here, but mention, without proof, that all basic components of Figure 5.0 are DI components.

EXAMPLE 6.3.2. Since the TOGGLE and XOR component are DI components, we conclude by Theorem 6.3.1, that the decomposition of the modulo-3 counter in Example 6.0.1 is a DI decomposition.

Because the SEQ, XOR, and WIRE components are DI components, it follows by Theorem 6.3.1 that the decomposition of the token-ring interface given in Example 6.1.3 is also a DI decomposition.

The specification of an OR gate given in Example 6.0.3 is not a DI component. (Notice that there can be two inputs c in a row, which is not allowed for a DI component.) In general, this may be an indication that the decomposition is not a DI decomposition. From Example 6.0.4 it followed that indeed the decomposition of Example 6.0.3 is not a DI decomposition.
□

7. CONCLUDING REMARKS

In this paper we have described a formal approach to the design of delay-insensitive circuits. We have specified circuits, or components as we call them formally, by means of regular expression-like programs representing all communication behaviors between component and

environment. Subsequently, such a program is manipulated by means of a calculus into a set of programs representing basic components. The connection of these basic components forms a realization of the desired component such that the functional operation of the connection is insensitive to component delays. Such a realization is called a decomposition. If, furthermore, all components involved are so-called DI components, then we may even conclude that this connection is insensitive to delays in components *and* connection wires, i.e. the decomposition is also a DI decomposition. DI components can be characterized as components whose communication behavior with the environment is insensitive to wire delays. We have illustrated this formal approach to circuit design by means of several examples including a modulo-3 counter and a token-ring interface.

Although we have covered only a few topics of [5], we hope we have revealed some of the aspects of the fascinating and many-sided research on the design of delay-insensitive circuits. We name a few of these aspects:

Language design: which programming primitives do we include in the language in order to be able to express a component in a clear and concise program? Here, for example, we have used the language of regular expressions and extended it with the operations weaving and projection in order to allow succinct expressions for components. It is also possible to include tail recursion in the language in order to be able to express finite state machine tersely.

Programming methodology: how do we design programs (i.e. commands) for components from given specifications ? Although we have not used it explicitly here, there exists a technique called the conjunction-weave rule with which parallel programs can be derived in a natural way from a specification [9, 18].

Translation techniques: how do we translate programs into connections of basic elements? The translations we have given in this paper are based on the definition of decomposition and a few useful theorems that could be formulated on decomposition. We believe that there exist many more theorems on decomposition that may be helpful in finding, or even deriving, decompositions in a constructive way. By developing a calculus on decomposition we may thus obtain translation techniques based on the syntax of commands, for example.

Syntax and semantics: how can we can satisfy *semantic* properties (like being a DI component or being a decomposition) by imposing *syntactic* requirements on programs? For the semantic property of 'being a DI component' we have developed a so-called *DI grammar,* i.e. a grammar that generates commands representing DI components. For decomposition we can formulate theorems, like the Separation Theorem, which depend on the syntax of a command.

VLSI design: what physical constraints must be met in order to implement the circuit designs obtained in the VLSI medium?

The most important results reported in [5] can be phrased briefly as follows. It is shown that any DI component can be decomposed into a finite basis of DI components. This basis essentially consists of the components given in Figure 5.0. Moreover, it is shown that if a component is specified by a command satisfying a specific DI grammar, then it can be decomposed into a number of basic components that is linear in the length of the command.

Finally, we want to emphasize that in approach presented our first concern has been the correctness of the designs and only in the second place have we addressed their efficiency.

Although the results are theoretically interesting, we believe that still many improvements can be made in translating programs into delay-insensitive circuits to make it also a practically interesting design method.

ACKNOWLEDGEMENTS

Martin Rem, Huub Schols, and the members of the Eindhoven VLSI Club are gratefully acknowledged for their suggestions of improvement on earlier drafts of this paper. I am especially indebted to Charles Molnar for his careful reading and many valuable comments.

REFERENCES

[1] T.J. CHANEY, *A Comprehensive Bibliography on Synchronizers and Arbiters,* Technical Memorandum No. 306C, Institute for Biomedical Computing, Washington University, St. Louis.

[2] T.J. CHANEY and C.E. MOLNAR, Anomalous Behavior of Synchronizer and Arbiter Circuits, *IEEE Transactions on Computers,* Vol. C-22 (1973), pp. 421-422.

[3] W.A. CLARK, Macromodular Computer Systems, *Proceedings of the Spring Joint Computer Conference,* AFIPS, April 1967.

[4] W.A. CLARK and C.E. MOLNAR, Macromodular Computer Systems, *Computers in Biomedical Research,* Vol. IV, (R. STACY, and B. WAXMAN eds.), Academic Press, New York, 1974.

[5] JO C. EBERGEN, *Translating Programs into Delay-insensitive Circuits,* Ph. D. Thesis, Eindhoven University of Technology, 1987.

[6] C.A.R. HOARE, Communicating Sequential Processes, *Communications of the ACM,* **21** (1978), pp. 666-677.

[7] C.A.R. HOARE, *Communicating Sequential Processes,* Prentice-Hall, 1985.

[8] M. HURTADO, *Dynamic Structure and Performance of Asymptotically Bistable Systems,* D. Sc. Dissertation, Washington University, St. Louis, 1975.

[9] ANNE KALDEWAIJ, *A Formalism for Concurrent Processes,* Ph.D. Thesis, Eindhoven University of Technology, 1986.

[10] ANNE KALDEWAIJ, The Translation of Processes into Circuits, in: *Proceedings PARLE, Parallel Architectures and Languages Europe,* Vol. 1, (J.W. DE BAKKER, A.J. NIJMAN and P.C. TRELEAVEN eds.), Springer-Verlag, 1987, pp. 195-213.

[11] ZVI KOHAVI, *Switching and Finite Automata Theory,* McGraw-Hill, 1970.

[12] ALAIN J. MARTIN, The Design of a Self-Timed Circuit for Distributed Mutual Exclusion, *Proceedings 1985 Chapel Hill Conference on VLSI,* (H. FUCHS ed.), Computer Science Press, 1985, pp. 247-260.

[13] ALAIN J. MARTIN, Compiling Communicating Processes into Delay-Insensitive VLSI Circuits, *Distributed Computing,* **1** (1986), pp. 226-234.

[14] CARVER MEAD and MARTIN REM, Minimum Propagation Delays in VLSI, *IEEE Journal of Solid-State Circuits,* Vol. SC-17 (1982), pp. 773-775.

[15] R.E. MILLER, *Switching Theory,* Wiley, 1965.

[16] R.E. MILLER, Chapter 10 in: [15], Vol. 2.

[17] C.E. MOLNAR, T.P. FANG and F.U. ROSENBERGER, Synthesis of Delay-Insensitive Modules, *Proceedings 1985, Chapel Hill Conference on VLSI*, (H. FUCHS ed.), Computer Science Press, 1985, pp.67-86.

[18] MARTIN REM, Concurrent Computations and VLSI Circuits, in: *Control Flow and Data Flow: Concepts of Distributed Computing*, (M. BROY ed.), Springer-Verlag, 1985, pp. 399-437.

[19] MARTIN REM, Trace Theory and Systolic Computations, *Proceedings PARLE, Parallel Architectures and Languages Europe*, Vol. 1, (J.W. DE BAKKER, A.J. NIJMAN and P.C. TRELEAVEN eds.), Springer-Verlag, 1987, pp. 14-34.

[20] HUUB M.J.L. SCHOLS, *A Formalisation of the Foam Rubber Wrapper Principle*, Master's Thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1985.

[21] HUUB M.J.L. SCHOLS and TOM VERHOEFF, *Delay-Insensitive Directed Trace Structures Satisfy the Foam Rubber Wrapper Postulate*, Computing Science Notes 85/04, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1985.

[22] C.L. SEITZ, System Timing, in: CARVER MEAD AND LYNN CONWAY, *Introduction to VLSI Systems*, Addison-Wesley, 1980, pp. 218-262.

[23] CLAUDE E. SHANNON, A Symbolic Analysis of Relay and Switching Circuits, *Trans. AIEE*, **57**, 1938, pp. 723-731.

[24] JAN L.A. VAN DE SNEPSCHEUT, *Trace Theory and VLSI Design*, LNCS 200, Springer-Verlag, 1985.

[25] ALAN M. TURING, Lecture to the London Mathematical Society on 20 February 1947, in: *Charles Babbage Institute Reprint Series for the History of Computing*, Vol. 10, (B.E. CARPENTAR and R.W. DORAN eds.), MIT Press, 1986.

[26] JAN TIJMEN UDDING, *Classification and Composition of Delay-Insensitive Circuits*, Ph.D. Thesis, Eindhoven University of Technology, 1984.

[27] JAN TIJMEN UDDING, A Formal Model for Defining and Classifying Delay-Insensitive Circuits and Systems, *Distributed Computing*, **1** (1986), pp. 197-204.

[28] S.H. UNGER, *Asynchronous Sequential Switching Circuits*, Wiley Interscience, 1969.

In this series appeared :

No.	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
86/04	G.J. Houben J. Paredaens K.M. van Hee	The partition of an information system in several parallel systems
86/05	Jan L.G. Dietz Kees M. van Hee	A framework for the conceptual modeling of discrete dynamic systems
86/06	Tom Verhoeff	Nondeterminism and divergence created by concealment in CSP
86/07	R. Gerth L. Shira	On proving communication closedness of distributed layers
86/08	R. Koymans R.K. Shyamasundar W.P. de Roever R. Gerth S. Arun Kumar	Compositional semantics for real-time distributed computing (Inf.&Control 1987)
86/09	C. Huizing R. Gerth W.P. de Roever	Full abstraction of a real-time denotational semantics for an OCCAM-like language
86/10	J. Hooman	A compositional proof theory for real-time distributed message passing
86/11	W.P. de Roever	Questions to Robin Milner - A responder's commentary (IFIP86)
86/12	A. Boucher R. Gerth	A timed failures model for extended communicating processes

86/13	R. Gerth W.P. de Roever	Proving monitors revisited: a first step towards verifying object oriented systems (Fund. Informatica IX-4)
86/14	R. Koymans	Specifying passing systems requires extending temporal logic
87/01	R. Gerth	On the existence of sound and complete axiomatizations of the monitor concept
87/02	Simon J. Klaver Chris F.M. Verberne	Federatieve Databases
87/03	G.J. Houben J.Paredaens	A formal approach to distri-buted information systems
87/04	T.Verhoeff	Delay-insensitive codes - An overview
87/05	R.Kuiper	Enforcing non-determinism via linear time temporal logic specification.
87/06	R.Koymans	Temporele logica specificatie van message passing en real-time systemen (in Dutch).
87/07	R.Koymans	Specifying message passing and real-time systems with real-time temporal logic.
87/08	H.M.J.L. Schols	The maximum number of states after projection.
87/09	J. Kalisvaart L.R.A. Kessener W.J.M. Lemmens M.L.P. van Lierop F.J. Peters H.M.M. van de Wetering	Language extensions to study structures for raster graphics.
87/10	T.Verhoeff	Three families of maximally nondeter-ministic automata.
87/11	P.Lemmens	Eldorado ins and outs. Specifications of a data base management toolkit according to the functional model.
87/12	K.M. van Hee and A.Lapinski	OR and AI approaches to decision support systems.
87/13	J.C.S.P. van der Woude	Playing with patterns, searching for strings.
87/14	J. Hooman	A compositional proof system for an occam-like real-time language

87/15	C. Huizing R. Gerth W.P. de Roever	A compositional semantics for statecharts
87/16	H.M.M. ten Eikelder J.C.F. Wilmont	Normal forms for a class of formulas
87/17	K.M. van Hee G.-J.Houben J.L.G. Dietz	Modelling of discrete dynamic systems framework and examples
87/18	C.W.A.M. van Overveld	An integer algorithm for rendering curved surfaces
87/19	A.J.Seebregts	Optimalisering van file allocatie in gedistribueerde database systemen
87/20	G.J. Houben J. Paredaens	The R^2 -Algebra: An extension of an algebra for nested relations
87/21	R. Gerth M. Codish Y. Lichtenstein E. Shapiro	Fully abstract denotational semantics for concurrent PROLOG
88/01	T. Verhoeff	A Parallel Program That Generates the Möbius Sequence
88/02	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable Specification for Information Systems
88/03	T. Verhoeff	Settling a Question about Pythagorean Triples
88/04	G.J. Houben J.Paredaens D.Tahon	The Nested Relational Algebra: A Tool to handle Structured Information
88/05	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable Specifications for Information Systems
88/06	H.M.J.L. Schols	Notes on Delay-Insensitive Communication
88/07	C. Huizing R. Gerth W.P. de Roever	Modelling Statecharts behaviour in a fully abstract way
88/08	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	A Formal model for System Specification
88/09	A.T.M. Aerts K.M. van Hee	A Tutorial for Data Modelling

88/10 J.C. Ebergen A Formal Approach to Designing Delay Insensitive Circuits