Distrib Comput (1991) 5:107–119

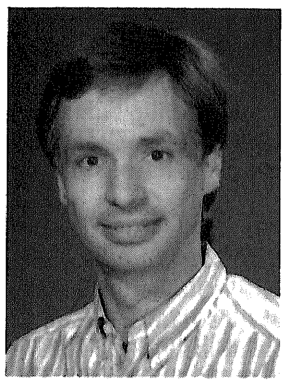
DISTRIBUTED COMPUTING
© Springer-Verlag 1991

A formal approach to designing delay-insensitive circuits

Jo C. Ebergen*

Computer Science Department, University of Waterloo, Waterloo Ontario, Canada N2L 3G1

Received June 1988 / Accepted July 1991



Jo C. Ebergen received his Master's degree in Mathematics from Eindhoven University of Technology in 1983. From 1983 until 1987 he has been working as a researcher at the Centre for Mathematics and Computer Science in Amsterdam in the area of VLSI design. In 1987, he received his Ph.D. degree from Eindhoven University of Technology. Currently, he is assistant professor at the University of Waterloo. His main research interests are programming methodology, parallel computations, and delay-insensitive circuit design. Dr. Ebergen is a member of ACM and EATCS.

Summary. 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 network of basic elements realizing the specified circuit. The notion of delay-insensitivity is concisely formalized.

Key words: Delay-insensitive circuit – Asynchronous circuit – Trace semantics – Communication behaviors – Regular expressions

1 Introduction

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 network of basic elements of which the correctness is insensitive to delays in basic elements and connection wires. The principal idea of our approach is to construct such a network 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 network of these elements forms a realization of the desired circuit.

The formal techniques and examples presented here form an extract from [8]. 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 network of elements chosen from a finite set of basic elements. Moreover, this translation has the property that the number of basic elements in the network is proportional to the length of the program.

In this paper we give a short introduction to the formalism presented in [8] 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 [9, 10] and developed at the University of Technology in Eindhoven for reasoning about parallel computations [23, 24, 27] and delay-insensitive circuits [12, 25, 30, 31, 33].

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 for expressing parallelism and projection (for hiding internal symbols).

Based on trace theory the concepts of *decomposition*

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

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 network of other components such that the correctness of the network 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 network of components such that the correctness of the network 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 circuit. 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 *asynchonous circuits*.

Synchronous circuits are circuits that perform their computations in lockstep with the successive pulses of a 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 [29]:

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 a difficult subject. Several techniques for the design of such circuits have been developed and are discussed in, for example, [13, 18, 32]. For special types of such circuits, formalizations and other design techniques have been proposed and discussed. 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 [19]. Informally speaking, speed-independent circuits are characterized as circuits of which the correctness is insensitive to element delays.

From a design discipline that was developed as part of the Macromodules project [5, 6] 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 of which the correctness 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' property, was later given in [30].

Another name that is frequently used in the design of asynchronous circuits is *self-timed systems*. This name has been introduced by Seitz [26] 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, Martin has proposed some interesting and promising design techniques for speed-independent circuits [14, 15]. His techniques are based on the compilation of CSP-like programs into networks of basic elements. The techniques presented in [8] exhibit a similarity with the techniques applied by Martin in the sense that they are both aimed at the translation of programs into networks 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 [26].

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 [3, 16]. 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. [2]). 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 [3, 11] due to the pioneering work on this phenomenon at 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 can be made increasingly smaller. It turns 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 [17, 26]. 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 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 ([27]).

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 networks 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 network 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 be smaller than the clock period.

4 Trace structures and commands

Trace theory is used as the underlying formalism for the design of delay-insensitive circuits. Components, i.e., abstractions of circuits, are specified by so-called *trace structures* satisfying certain properties. We briefly explain what trace structures are and how they can be denoted in the same way as regular expressions denote regular languages.

A *trace structure* is a triple $\langle A, B, X\rangle$, 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 $\emptyset^* = \{\varepsilon\}$. For a trace structure $R = \langle A, B, X\rangle$, 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, 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 lower-case letters with exception of r, s, and t.

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

$$\begin{array}{lll}
R;S = \langle \mathbf{i}R \cup \mathbf{i}S, & \mathbf{o}R \cup \mathbf{o}S, & (\mathbf{t}R)(\mathbf{t}S)\rangle, \\
R|S = \langle \mathbf{i}R \cup \mathbf{i}S, & \mathbf{o}R \cup \mathbf{o}S, & \mathbf{t}R \cup \mathbf{t}S\rangle, \\
*[R] = \langle \mathbf{i}R, & \mathbf{o}R, & (\mathbf{t}R)^*\rangle, \\
\mathbf{pref}R = \langle \mathbf{i}R, & \mathbf{o}R, & \{t_0 | (\exists t_1 :: t_0 t_1 \in \mathbf{t}R)\}\rangle, \\
R\downarrow A = \langle \mathbf{i}R \cap A, & \mathbf{o}R \cap A, & \{t\downarrow A | t \in \mathbf{t}R\}\rangle, \quad \text{and} \\
R \| S = \langle \mathbf{i}R \cup \mathbf{i}S, & \mathbf{o}R \cup \mathbf{o}S, & \\
 & \multicolumn{2}{l}{\{t \in (\mathbf{a}R \cup \mathbf{a}S)^* | t\downarrow \mathbf{a}R \in \mathbf{t}R \wedge t\downarrow \mathbf{a}S \in \mathbf{t}S\}\rangle,}
\end{array}$$

where $t\downarrow C$ denotes the projection of trace t on alphabet C, i.e., 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, non-empty trace structures for the specification of components. A trace structure R is called non-empty if $\mathbf{t}R \neq \emptyset$.

The projection operator allows us to hide 'internal' symbols.

The weave operator 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. For example, we have

$$\langle\{a\},\{c\},\{ac\}\rangle \| \langle\{b\},\{c\},\{bc\}\rangle = \langle\{a,b\},\{c\},\{abc, bac\}\rangle.$$

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

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

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

Notational convention. In the following, commands are denoted by the capital E. The input and output alphabet and the trace set of the 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, weaving has highest priority, then concatenation, and finally union.

Property 1. *Every command represents a regular trace structure.*

Example 1. Syntactically different commands can express the same trace structure. We have, for example,

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

5 Specifications and their interpretation

Components are specified by trace structures satisfying certain properties. In this paper we shall keep to regular components, i.e., to regular trace structures. We explain how a trace structure prescribes all possible communication behaviors between a component and environment at their mutual boundary.

The set of all communication behaviors between a component and its environment is specified by a prefix-closed, non-empty, trace structure R with $\mathbf{i}R \cap \mathbf{o}R = \emptyset$. The alphabet of R consists of the names of all terminals at which component and environment communicate with each other. This set of terminals is called the *boundary* between component and environment. 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. Because $\mathbf{i}R \cap \mathbf{o}R = \emptyset$, a communication action may be produced either by the component or by the environment. Furthermore, a communication action is always produced by the same communicant.

A communication behavior evolves by the production of communication actions. A production of a communication action at a terminal is represented by the name of that terminal. The set $\mathbf{t}R$ specifies which communication action may be produced for each sequence of communication actions thus far. Let the communication actions that have already taken place be the trace $t \in \mathbf{t}R$, and let $tb \in \mathbf{t}R$. (Initially, $t = \varepsilon$.) If $b \in \mathbf{i}R$, then the environment is allowed to 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 2. Consider the command E given by $E = \mathbf{pref}*[a? \| b?; c!]$. This command specifies a component for which the environment initially is allowed to produce 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 may repeat.

With the interpretation above of a specification we explicitly prescribe how the component may behave and how the environment may communicate with the component. Later, when we are interested in realizing components by networks of other components, we assume that the environment of this network behaves as prescribed. Under this assumption the network has to react as prescribed for the component.

In the above we have used the phrase 'may produce' several times. This phrase can be interpreted in different ways. An obvious interpretation of 'a component may produce output b' is that the component can choose whether to produce output b or not. One possible choice could be 'never produce output b'. If we adopt this interpretation, implementing a component in such a way that it behaves according to its specification is simple. A universal 'accept-everything-do-nothing' component would do the job, since it accepts every input that the environment may produce and never chooses to produce an output. For example, the component $\mathbf{pref}*[a? \| b?; c!]$ then could be implemented by a component that accepts any sequence of a's and b's, but never produces a c.

In this paper, we adopt a slightly more restrictive interpretation of 'may produce'. We require that every trace specified is possible, though not guaranteed, to occur in an implementation. The actual occurrence of a trace depends on the choices made by the nondeterministic components, if any, i.e., every trace specified can be produced by an implementation if the right choices are made by the nondeterministic components. Adopting this interpretation of a specification, we say that an implementation *behaves as specified* if every trace is possible to occur in an implementation. Notice that this interpretation excludes the implementation of $\mathbf{pref}*[a? \| b?; c!]$ by an 'accept-everything-do-nothing' component, since then trace abc can not occur. We shall return to this interpretation of a specification when we formulate the conditions for a decomposition of a component.

Given a trace structure R specifying a communication behavior between a component and an environment, we often speak of component R and environment R. When speaking of component R, the inputs of R are interpreted as the component's inputs and the outputs of R as the component's outputs. When speaking of environment R, the inputs of R are interpreted as the environment's outputs and the outputs of R as the environment's inputs. The role of component and environment can be interchanged by *reflecting* R:

Definition 2. The reflection of R, denoted by $\bar{R}$, is defined by $\bar{R} = \langle \mathbf{o}R, \mathbf{i}R, \mathbf{t}R \rangle$.

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

Without proof we mention that the reflection of a command E can be obtained simply by replacing atomic commands $a!$ and $a?$ in E by $a?$ and $a!$ respectively. For example, if E is given by $E = \mathbf{pref}*[a? \| b?; c!]$, then $\bar{E} = \mathbf{pref}*[a! \| b!; c?]$.

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.

6 Specifications of some components

In Fig. 1 a set of components is specified by means of commands. The first column lists the names of the components, the second column the specifications, and the third column the schematics.

The WIRE and IWIRE describe the transmission of a signal from terminal to terminal. Both components have the same behavior except for a difference in initial states. For the WIRE $\mathbf{pref}*[a?; b!]$, the environment initially is allowed to produce input a. For the IWIRE $\mathbf{pref}*[b!; a?]$, the component initially may produce 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 WIRE corresponds to a physical wire. Notice that, there is always at most one transition propagating along a WIRE or IWIRE as long as the environment behaves as prescribed in the specification.

A FORK performs the primitive operation of duplication of inputs.

A C-ELEMENT performs the primitive operation of synchronization on an output. (Recall from Example 1 that a C-ELEMENT and a FORK can be represented by other commands.) A C-ELEMENT is a reflection of a FORK, apart from a different initialization. (The C-ELEMENT can be implemented by the Muller C-element, named after David E. Muller.)

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

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

The SEQUENCER is a kind of arbiter. For a SEQUENCER 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 in the trace set of the SEQUENCER, 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 SEQUENCER may grant one request for each occurrence of input n. In sequencing the grants it may have to arbitrate among two 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 may be produced.

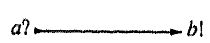
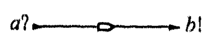
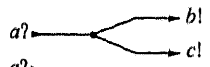
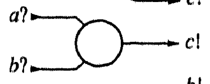
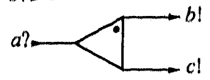
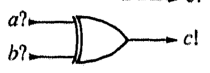
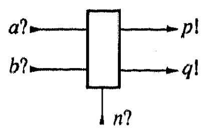
Name	Specification	Schematic
WIRE	$\mathbf{pref}*[a?; b!]$	
IWIRE	$\mathbf{pref}*[b!; a?]$	
FORK	$\mathbf{pref}*[a?; b! \| c!]$	
C-ELEMENT	$\mathbf{pref}*[a? \| b?\ c!]$	
TOGGLE	$\mathbf{pref}*[a?; b!; a?; c!]$	
MERGE	$\mathbf{pref}*[(a? \| b?); c!]$	
SEQUENCER	$\mathbf{pref}*[a?; p!]$ $\| \mathbf{pref}*[b?; q!]$ $\| \mathbf{pref}*[n?; (p! \| q!)]$	

Fig. 1. Specifications of some components

7 Two specification examples

Example 3. 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 $n \bmod 3 \neq 0$, output q may be produced; if $n \bmod 3 = 0$, then output p may be produced. This behavior is expressed in command $E0$, where

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

Notice that the TOGGLE of Fig. 1 can be considered as a modulo-2 counter. In a later section we discuss

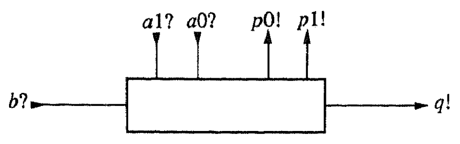


Fig. 2. A token-ring interface

a decomposition of the modulo-3 counter into components of Fig. 1. (Before reading that section, the reader may try to find such a network.)

Example 4. 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 ringwise 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 Fig. 2. 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 specification $\mathbf{pref}*[a1?;p1!;a0?;p0!]$.

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 specification $\mathbf{pref}*[b?;q!]$.

The synchronization between the two protocols must satisfy the following requirements. After each receipt of the token, the token may either be sent on to the next token-ring interface or, if there is also a request from the machine, the token may 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 command

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

The first line represents the communication behavior between machine and token-ring interface; the second line represents the 'flow' of the token. By weaving these two specifications, we establish the proper synchronization: since weaving denotes synchronization on common symbols, alternative $p1!;a0?;q!$ of the second line may be executed only if $p1!$ in the first line can be executed as well. Consequently, a token is only sent to the machine if both a token *and* a request have been received. In the following sections we show how this synchronization can be realized by a delay-insensitive network of basic elements.

8 Decomposition

The definition of *decomposition*, which we are about to formulate, formalizes the idea of 'realizing a component by a network of (other) components such that the correctness of the network is insensitive to delays in the components'.

We first present the definition of decomposition and then give a brief motivation for it. Whenever we speak of a component S in the following, we mean that S is a prefix-closed, non-empty trace structure with disjoint input and output alphabet.

Definition 3. Let $n>1$. We say that component $S.0$ can be decomposed into the components $S.i$, $1\leq i<n$, denoted by

$$S.0 \longrightarrow (i{:}1\leq i<n{:}S.i),$$

if the following conditions are satisfied. Let $R.0=\overline{S.0}$, $R.i=S.i$ for $1\leq i<n$, and $W=(\|i{:}0\leq i<n{:}R.i)$.
(i) Closed network:
$(\cup i{:}0\leq i<n{:}\mathbf{o}(R.i))=(\cup i{:}0\leq i<n{:}\,\mathbf{i}(R.i))$.
(ii) No output interference:
$\mathbf{o}(R.i)\cap\mathbf{o}(R.j)=\emptyset$ for $0\leq i,j<n$ and $i\neq j$.
(iii) Network is free of computation interference:
For all traces t, symbols x, and indexes i, where $0\leq i<n$, we have

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

(iv) Network behaves as specified at boundary $\mathbf{a}(R.0)$:
$\mathbf{t}W\!\downarrow\!\mathbf{a}(R.0)=\mathbf{t}(R.0)$.

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, $S.0\longrightarrow(i{:}1\leq i<4{:}S.i)$, we sometimes write $S.0\longrightarrow(S.1, S.2, S.3)$. Here, the comma separates the components.

In Sect. 5, we stipulated that a trace structure $S.0$ prescribes both the behavior of component and environment: it specifies when the component may produce the outputs of $S.0$ and when the environment is allowed to produce the inputs of $S.0$. In a decomposition of component $S.0$ we require that the production of the outputs is realized by a network of components. We assume that the environment of this network produces the inputs as specified for environment $\underline{S.0}$. 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 network of components $S.i$, where $1 \leq i < n$, *and* $\overline{S.0}$.

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

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

Condition (iii) requires that every output that can be produced can also be accepted as input, i.e., no environment prescription is violated. If (iii) holds we say that the network is free of *computation interference*. Otherwise, we say that the network has danger of computation interference. Recall that $t \in \mathbf{t}W$ means that the joint behavior t is in accordance with the behavior of every component in the network. In words, condition (iii) says that for every trace t, symbol x, and component $R.i$, where $0 \leq i < n$ and after joint behavior t component $R.i$ can produce output x, the production of output x again results in a joint behavior tx from $\mathbf{t}W$.

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

Condition (iv) requires that the closed network behaves as specified by $\mathbf{t}(S.0)$ at the boundary $\mathbf{a}(S.0)$. According to our interpretation of a specification, every trace specified should be possible to occur. All possible behaviors of the network are given by $\mathbf{t}W = \mathbf{t}(\|i{:}0 \leq i < n{:}R.i)$. Restriction of this behavior to the boundary $\mathbf{a}(R.0)$ $(= \mathbf{a}(S.0))$ is expressed by $\mathbf{t}W \downarrow \mathbf{a}(R.0)$.

Condition (iv) does not require that after a certain trace an output is guaranteed to occur. It only requires that each trace in $\mathbf{t}E$ *may* occur in the simulation. The actual occurrence of a trace in a simulation depends on the choices made by the non-deterministic components. Consequently, conditions (i) through (iv) do not guarantee, for example, fairness nor absence of the danger of deadlock or livelock. If such additional conditions are required for a decomposition, they will have to be formulated. This is still a topic of further research. In other works on delay-insensitive circuits ([4, 7]) condition (iv) is not required to hold.

In the present paper, we take the four conditions above as our correctness criteria for a decomposition. They are simple to verify. An automatic verifier for conditions (i), (ii), and (iii) has been designed and is described in [7]. The time complexity of a straightforward verification algorithm, however, can be exponential in n, where n is the number of components in the network. Theorems that assist the designer in verifying or finding a decomposition in a possibly more efficient way are given in [7, 8]. We briefly discuss some of these theorems in a later section.

Notice that we have described decomposition as a goal-directed activity: we start with a component $S.0$ and try to find components $S.i$, $1 \leq i < n$, such that the relation $S.0 \longrightarrow (i{:}1 \leq i < n{:}S.i)$ holds. Thus, we explicitly use the assumption that the environment of the network of components behaves as specified for environment $S.0$. We do not start with components $S.i$, $1 \leq i < n$, to find out what could be made of them without requiring anything from the environment.

9 Decomposition examples

Example 5. We demonstrate that the modulo-3 counter $E0$ of Example 3 can be decomposed into a MERGE and two TOGGLES. A schematic of this decomposition is given in Fig. 3, where each symbol x that is an output of one component and an input of an other component is denoted by $!x?$. To verify this decomposition we take

$R.0 = \mathbf{pref}*[a!; q?; a!; q?; a!; p?]$,
$R.1 = \mathbf{pref}*[(a?|d?); b!]$,
$R.2 = \mathbf{pref}*[b?; q!; b?; c!]$, and
$R.3 = \mathbf{pref}*[c?; d!; c?; p!]$,

where $R.0 = \overline{E0}$. By inspection, we observe that the network of the components $R.0$, $R.1$, $R.2$, and $R.3$ is closed and free of output interference.

Furthermore, we have

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

Each symbol x in the command for W is denoted by $!x?$, since x is an output of one component and an input of some other component. This also holds for the symbols a, p, and q, because the environment is taken into account in $R.0$.

From the expression for $\mathbf{t}W$ we derive that $\mathbf{t}W \downarrow \mathbf{a}(R.0) = \mathbf{t}(R.0)$. (Notice that in a trace set, the direction of a symbol does not play a role.) Accordingly, we conclude that the network behaves as specified at the boundary $\mathbf{a}(R.0)$.

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

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

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

$$t \in \mathbf{t}W \wedge x \in \mathbf{o}(R.i) \wedge tx \downarrow \mathbf{a}(R.i) \in \mathbf{t}(R.i). \tag{1}$$

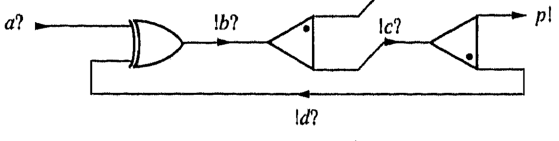


Fig. 3. A decomposition of the modulo-3 counter

(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 automation 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 (1) indeed $tx \in \mathbf{t}W$. Consequently, we conclude that Fig. 3 gives a decomposition of the modulo-3 counter.

Example 6. Similar to Example 5 we can prove the decompositions

$$\begin{aligned}&\mathbf{pref}*[a1?;p1!;a0?;p0!]\\ \longrightarrow&(\mathbf{pref}*[a1?;p1!], \mathbf{pref}*[a0?;p0!]) \qquad (2)\end{aligned}$$

$$\begin{aligned}&\mathbf{pref}*[b?;(q!|p1!;a0?;q!)]\\ \longrightarrow&(\mathbf{pref}*[b?;(q1!|p1!;a0?;q0!)] \qquad (3)\\ &, \mathbf{pref}*[(q1?|q0?);q!]\\ &)\end{aligned}$$

$$\begin{aligned}&\mathbf{pref}*[b?;(q1!|p1!;a0?;q0!)]\\ \longrightarrow&(\mathbf{pref}*[b?;(q1!|p1!)] \qquad (4)\\ &, \mathbf{pref}*[a0?;q0!]\\ &).\end{aligned}$$

Decompositions (2) and (4) are simple. In decomposition (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 MERGE these two symbols can then the merged again into output q.

Example 7. 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

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

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

Component $E2$ is a C-ELEMENT 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 Fig. 4, where we have taken the schematic of the OR gate as the schematic for component $E1$. In Fig. 4 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 in 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 8. Suppose that we would consider the fork in Fig. 4 of the previous example as a genuine FORK in a tentative decomposition of $E0$. The schematic of this tentative decomposition is given in Fig. 5. We demonstrate that this tentative decomposition does not satisfy all conditions: it has danger of computation interference. Let

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

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

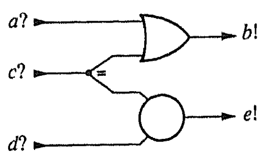


Fig. 4. A decomposition of $E0$

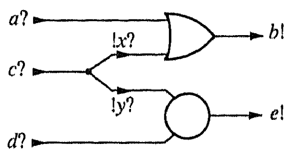


Fig. 5. A tentative decomposition of $E0$

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

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

Then $R.0 = \overline{E0}$. 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.,

$t \in \mathbf{t}W \wedge a \in \mathbf{o}(R.0) \wedge ta \downarrow \mathbf{a}(R.0) \in \mathbf{t}(R.0)$.

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 network 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 $E0$ as given in the previous example, the fork in Fig. 4 must be implemented by a so-called *isochronic fork* [14, 15]. As 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.

10 Two theorems on decomposition

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

The Substitution 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 $S.0$?

Theorem 4 (Substitution Theorem). *For components $S.0$, $S.1$, $S.2$, $S.3$, and T we have*

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

if the following condition is satisfied.

$$(\mathbf{a}(S.0) \cup \mathbf{a}(S.1)) \cap (\mathbf{a}(S.2) \cup \mathbf{a}(S.3)) = \mathbf{a}T. \qquad (5)$$

The proof of this theorem can be found in [8] (pp. 49–51).

Condition (5) states that the only symbols that the decompositions of $S.0$ and T have in common are symbols from $\mathbf{a}T$. It 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 $(\mathbf{a}(S.2) \cup \mathbf{a}(S.3)) \backslash \mathbf{a}T$, where '$\backslash$' means set deletion.

The theorem above 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.

$$\begin{aligned}
& \quad S.0\\
\longrightarrow\ & \{\text{hint why } S.0 \longrightarrow (S.1, T)\}\\
& \quad (S.1, T)\\
\longrightarrow\ & \{\text{hint why } T \longrightarrow (S.2, S.3)\}\\
& \quad (S.1, S.2, S.3).
\end{aligned}$$

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

Example 9. Applying the Substitution Theorem to Example 6, we derive

$$\begin{aligned}
& \quad \mathbf{pref}*[b?;(q!\,|\,p1!;a0?;q!)]\\
\longrightarrow\ & \{(3)\}\\
& \quad (\mathbf{pref}*[b?;(q1!\,|\,p1!;a0?;q0!)],\\
& \quad \mathbf{pref}*[(q1?\,|\,q0?);q!]\\
& \quad)\\
\longrightarrow\ & \{(4)\}\\
& \quad (\mathbf{pref}*[b?;(q1!\,|\,p1!)], \mathbf{pref}*[a0?;q0!],\\
& \quad \mathbf{pref}*[(q1!\,|\,q0?);q!]\\
& \quad).
\end{aligned}$$

Another theorem that is convenient in finding a decomposition is the Separation Theorem.

Theorem 5 (Separation Theorem). *For components $S.i$ and $T.i$, $0 \le i < n$, we have*

$$\begin{aligned}
& S.0 \qquad \longrightarrow (i:1 \le i < n: S.i)\\
& \wedge\ T.0 \qquad \longrightarrow (i:1 \le i < n: T.i)\\
\Rightarrow\ & S.0 \,\|\, T.0 \longrightarrow (i:1 \le i < n: S.i \,\|\, T.i)
\end{aligned}$$

if the following conditions are satisfied.

$$A \cap B = \emptyset \textit{ and} \qquad (6)$$

$$Out.i \cap Out.j = \emptyset \textit{ for } 0 \le i, j < n \wedge i \neq j. \qquad (7)$$

where

$$\begin{aligned}
& A = (\cup i:1 \le i < n: \mathbf{a}(S.i)) \backslash \mathbf{a}(S.0),\\
& B = (\cup i:1 \le i < n: \mathbf{a}(T.i)) \backslash \mathbf{a}(T.0),\\
& Out.i = \mathbf{o}(S.i) \cup \mathbf{o}(T.i) \textit{ for } 1 \le i < n, \textit{ and}\\
& Out.0 = \mathbf{o}(\overline{S.0}) \cup \mathbf{o}(\overline{T.0}).
\end{aligned}$$

The proof of this theorem can be found in [8] (pp. 53 55).

Condition (6) requires that the internal symbols of the decompositions of $S.0$ and $T.0$ are disjoint, where the sets of internal symbols for the decompositions of $S.0$ and $T.0$ are given by A and B respectively. Condition (6) can always be satisfied by an appropriate renaming of the internal symbols. Condition (7) requires that the outputs are 'column-wise' disjoint, where the outputs of 'column' i are given by $Out.i$ for $0 \le i < n$. Recall from the definition of decomposition that the ordering of the components in $(i:1 \le i < n: S.i)$ is not relevant. Accordingly, we may reorder the components such that condition (7) can be satisfied.

The Separation Theorem can be generalized in a natural way to decompositions of components that are expressed as weaves of more than two trace structures.

Example 10. We apply the Substitution and Separation Theorem to obtain a decomposition of the token-ring interface. The token-ring interface was specified by

$$\begin{aligned}
& \mathbf{pref}*[a1?;p1!;a0?;p0!]\\
\|\ & \mathbf{pref}*[b?;(q!\,|\,p1!;a0?;q!)].
\end{aligned}$$

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

$$\begin{aligned}
& E0 = \mathbf{pref}*[a1?;p1!;a0?;p0!] \text{ and}\\
& E1 = \mathbf{pref}*[b?;(q!\,|\,p1!;a0?;q!)].
\end{aligned}$$

From Examples 6 and 9 we derive that $E0$ and $E1$ can be decomposed as follows.

$$\begin{aligned}
E0 \longrightarrow\ & (\mathbf{pref}*[a1?;p1!]\\
& ,\mathbf{pref}*[a0?;p0!]\\
& ,\varepsilon\\
& ,\varepsilon\\
&)\\
E1 \longrightarrow\ & (\mathbf{pref}*[b?;(q1!\,|\,p1!)]\\
& ,\varepsilon\\
& ,\mathbf{pref}*[a0?;q0!]\\
& ,\mathbf{pref}*[(q1?\,|\,q0?);q!]\\
&).
\end{aligned}$$

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

Since $\emptyset \cap \{q0, q1\} = \emptyset$, the internal symbols of the decompositions of $E0$ and $E1$ are disjoint. Furthermore, we observe that the outputs are 'column-wise' disjoint. Consequently, we conclude by the Separation Theorem that

$$
\begin{array}{ll}
& E0 \parallel E1 \\
\longrightarrow & \{\text{Decompositions above, Separation Theorem}\} \\
& (\mathbf{pref}*[a1?;p1!] \parallel \mathbf{pref}*[b?;(q1!|p1!)] \\
& ,\mathbf{pref}*[a0?;p0!] \parallel \varepsilon \\
& ,\varepsilon \parallel \mathbf{pref}*[a0?;q0!] \\
& ,\varepsilon \parallel \mathbf{pref}*[(q1?|q0?);q!] \\
&).
\end{array}
$$

Since $E \parallel \varepsilon = E$, we may simplify the commands above even further.

In the list above, we recognize two WIRES and a MERGE. The command in the first line, however, is not a specification of a familiar component. This command almost looks like the command of a SEQUENCER. We are missing a part 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.

$$
\begin{array}{ll}
& \mathbf{pref}*[a1?;p1!] \parallel \mathbf{pref}*[b?;(q1!|p1!)] \\
\longrightarrow & \{\text{Def. of decomposition} \\
& \text{, introduction of internal symbol } rq1\} \\
& (\ \mathbf{pref}*[a1?;p1!] \\
& \ \parallel \mathbf{pref}*[rq1?;q1!] \\
& \ \parallel \mathbf{pref}*[b?;(q1!|p1!)] \\
& ,\mathbf{pref}*[rq1!;q1?] \\
&).
\end{array}
$$

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

$$
\begin{array}{ll}
& E0 \parallel E1 \\
\longrightarrow & \{\text{Decompositions above, Substitution Theorem}\} \\
& (\ \mathbf{pref}*[a1?;p1!] \\
& \ \parallel \mathbf{pref}*[rq1?;q1!] \\
& \ \parallel \mathbf{pref}*[b?;(q1!|p1!)] \\
& ,\mathbf{pref}*[rq1!;q1?] \\
& ,\mathbf{pref}*[a0?;p0!] \\
& ,\mathbf{pref}*[a0?;q0!] \\
& ,\mathbf{pref}*[(q1?|q0?);q!] \\
&).
\end{array}
$$

This decomposition is shown in Fig. 6. Since both $a0$ and $q1$ are inputs for two components, we have drawn in Fig. 6 two forks to depict the decomposition. Notice that these forks do not occur as genuine FORKS in the decomposition. Therefore, these forks have been drawn as isochronic forks. Later, in Example 11, we shall see, however, that these forks can be considered as genuine FORKS in the decomposition. That is, operationally speaking, arbitrary delays may take place in the branches of the forks without affecting the correctness of the decomposition.

11 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. Between the sub-circuits, however, it is assumed that the sending and receipt of a signal coincide, i.e., the communications are instantaneous. Thus, a circuit obtained by decomposition can be called a speed-independent circuit, i.e., its correctness is independent of any delays in the response times of the components.

In practice, the sub-circuits are connected by wires with unspecified delays. Then the assumption of instantaneous communications is clearly violated, unless we explicitly include the connection wires in the network of the sub-circuits. For this reason, we introduce WIRES in a DI decomposition. Instead of having direct connections between components, each component is connected to other components through one or two WIRES via an intermediate terminal. The set of intermediate terminals is called the *intermediate boundary* and is illustrated by a dashed line in Fig. 7.

Operationally speaking, WIRES introduce delays in the communications. Thus, they may affect the correctness of the decomposition. If, however, the closed network with WIRES is free of interference and behaves a specified, then we call such a network a delay-insensitive circuit.

The formalization of 'realizing a component by means of a delay-insensitive circuit' is done as follows. We first define the *enclosure* *enc(S.k)* of a component *S.k*, i.e., the component enclosed by the intermediate boundary, by renaming the symbols of component *S.k* to their 'localized' versions.

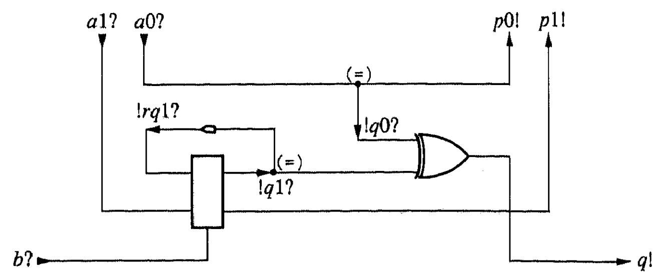


Fig. 6. A decomposition of the token-ring interface

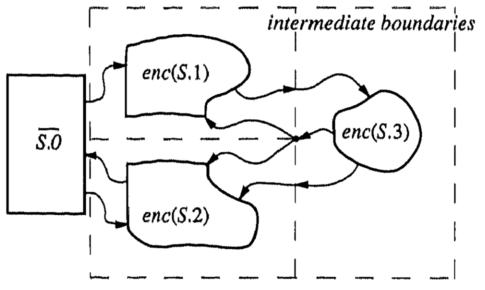


Fig. 7. DI decomposition

$enc(S.k)$ is the trace structure obtained by replacing
each output a in $S.k$ by $o\,a_k$ and
each input a in $S.k$ by $i\,a_k$.

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

$$Wire(k, a) = \mathbf{pref}*[o\,a_k?;\, a!] \text{ if } a \in \mathbf{o}(S.k)$$
$$= \mathbf{pref}*[a?;\, i\,a_k!] \text{ if } a \in \mathbf{i}(S.k).$$

The collection of all WIRES between $enc(S.k)$ and $\mathbf{a}(S.k)$ is denoted by $Wires(S.k)$. The definition of DI decomposition now reads as follows.

Definition 6. We say that the components $S.k$, $1 \leq k < n$, form a DI decomposition of component $S.0$, denoted by

$$S.0 \xrightarrow{\mathrm{DI}} (k\colon 1 \leq k < n\colon S.k),$$

if and only if

$$S.0 \longrightarrow (k\colon 1 \leq k < n\colon enc(S.k),\, Wires(S.k)).$$

Remark. The definition of DI decomposition given here is slightly different from the one given in [8], where the (closed) network is also required to be 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 [8]. The definitions are equivalent if $S.0$ is a DI component (cf. next section).

The name 'DI decomposition' suggests that it is a special case of decomposition. Unfortunately this is not true. There are DI decompositions that are not decompositions. Consider, for example, $E.0 = \mathbf{pref}(a! \,\|\, b!)$ and $E1 = \mathbf{pref}(a!;\, b!)$. Then $E0 \xrightarrow{\mathrm{DI}} (E1)$, but *not* $E0 \longrightarrow (E1)$. The problem here is that adding WIRES to a decomposition may enlarge the set of all possible behaviors. For example, two outputs that are ordered at a component's boundary may be unordered at the intermediate boundary, after adding WIRE extensions. In this respect, the name 'decomposition' is an unfortunate choice. A better name for decomposition would have been 'SI decomposition', since decomposition formalizes the notion of a speed-independent circuit. For a more detailed analysis of the possible definitions of decomposition and DI decomposition, we refer to [22].

12 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) WIRES. The two decompositions are equivalent, however, if all constituting components are so-called DI components. DI components are defined by

Definition 7. Component S is called a DI component, if

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

We have

Theorem 8. *If all components $S.i$, $1 \leq i < n$, are DI components, then*

$$S.0 \longrightarrow (i\colon 1 \leq i < n\colon S.i) \;\equiv\; S.0 \xrightarrow{\mathrm{DI}} (i\colon 1 \leq i < n\colon S.i).$$

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 part of the proof is more complicated. A complete proof can be found in [8].

Definition 7 formalizes that the possible communication behaviors between component and environment are insensitive to WIRE delays. The property $S \longrightarrow (Wires(S),\, enc(S))$ can be considered as a formalization of the so-called *Foam Rubber Wrapper* (FRW) property. Operationally speaking, the FRW property states that the specification of a component is invariant under the deformation of its boundary with connection WIRES. The boundaries of the 'Foam Rubber Wrapper' here are constituted by $\mathbf{a}\,enc(S)$ and $\mathbf{a}S$, as depicted in Fig. 8.

The idea of formalizing delay-insensitivity by means of the FRW property originates from Molnar [21]. Before the Foam Rubber Wrapper metaphore was coined, Fang had devised an incomplete set of Petri-Net rules in [20] that should be satisfied by any specification in order to comply with what later became the FRW property. Udding was the first to give a rigorous formulation of the FRW property in terms of trace structures. In [30] he postulates a number of rules that a component must satisfy in order to meet the FRW property. It turns out that Udding's definition of a DI component is equivalent to Definition 7 [8]. Schols [25], Dill [7], and Verhoeff [33] have given other equivalent characterizations of the FRW property.

In order to use Theorem 8 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 7, for example, or the rules given by 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 [8]. We shall not recapitulate these methods here, but mention, without proof, that all components of Fig. 1 are DI components.

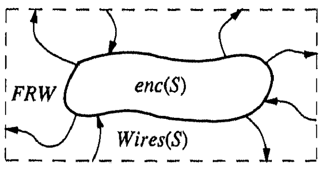


Fig. 8. S is a DI component: $S \longrightarrow (Wires(S),\, enc(S))$

Example 11. Since the WIRE, IWIRE, C-ELEMENT, TOGGLE, MERGE, and SEQUENCER, are DI components. We conclude, by Theorem 8, that the decompositions of the modulo-3 counter in Example 5 and the token-ring interface in Example 10 are a DI decompositions.

The specification of an OR gate given in Example 7 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. In Example 8 we demonstrated that there is danger of computation interference in case WIRES are introduced in the decomposition of Example 7. This essentially shows that the decomposition of Example 7 is not a DI decomposition.

13 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 network of these basic components forms a realization of the desired component such that the correctness 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 network 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 [8], we hope to 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 operation weaving, for example, 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 for components from given specifications? Although we have not used it explicitly here, there exists a technique called the conjunction-weave rule with which a parallel program can be derived in a natural way from a specification [23, 27].

Translation techniques: how do we translate programs into networks 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 more theorems on decomposition exist 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 programs, for example.

Syntax and semantics: how can we satisfy semantic properties (such as 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 a VLSI medium?

The most important results reported in [8] 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 Fig. 1. 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 the approach presented our first concern has been the correctness of the designs; only in the second place have we addressed their efficiency. Although these results are theoretically interesting, many other publications [1, 14, 28] have indicated that translating programs into delay-insensitive circuits has also become 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. Berkel CH van, Niessen C, Rem M, Saeijs R: VLSI programming and silicon complication: a novel approach from Philips Research. In: IEEE Computer Society, Proceedings of IEEE International Conference on Computer Design 1988. IEEE Computer Society Press, Washington DC 1988, pp 150–166
2. Chaney TJ: A comprehensive bibliography on synchronizers and arbiters. Technical Memorandum No 306C. Institute for Biomedical Computing, Washington University, St. Louis 1986
3. Chaney TJ, Molnar CE: Anomalous behavior of synchronizer and arbiter circuits. IEEE Trans Comput C-22: 421–422 (1973)
4. Chen W, Udding JT, Verhoeff T: Networks of communicating processes and their (de-)composition. In: van de Snepscheut JLA (ed) Mathematics of program construction. Lect Notes Comput Sci, vol 375. Springer, Berlin Heidelberg New York 1989, pp 174–196
5. Clark WA: Macromodular computer systems. In: AFIPS Proceedings of the spring joint computer conference. Academic Press London 1967, pp 335–401

6. Clark WA, Molnar CE: Macromodular computer systems. In: Stacy R, Waxman B (eds) Computers in biomedical research, vol IV. Academic Press, New York 1974, pp 45–85
7. Dill DL: Trace theory for automatic hierarchical verification of speed-independent circuits. MIT Press, Cambridge, Massachusetts 1989
8. Ebergen JC: Translating programs into delay-insensitive circuits. CWI Tract 56, Centre for Mathematics and Computer Science, Amsterdam 1989
9. Hoare CAR: Communicating sequential processes. Commun ACM 21:666–677 (1978)
10. Hoare CAR: Communicating sequential processes. Prentice-Hall, London 1985
11. Hurtado M: Dynamic structure and performance of asymptotically bistable systems. D. Sc. Dissertation, Washington University, St. Louis 1975
12. Kaldewaij A: The translation of processes into circuits. In: Bakker JW de, Nijman AJ, Treleaven PC (eds) Proceedings PARLE, parallel architectures and languages europe, vol 1. Springer, Berlin Heidelberg New York 1987, pp 195–213
13. Kohavi Z: Switching and finite automata theory. McGraw-Hill, New York 1970
14. Martin AJ: Programming in VLSI: from communicating processes to delay-insensitive circuits. In: Hoare CAR (ed) Developments in concurrency and communication. UT year of programming institute on concurrent programming. Addison-Wesley, Massachusetts 1989, pp 1–64
15. Martin AJ: Compiling communicating processes into delay-insensitive VLSI circuits. Distrib Comput 1:226–234 (1986)
16. Mead C, Conway L: Introduction to VLSI systems. Addison-Wesley, Reading, Massachusetts 1980
17. Mead C, Rem M: Minimum propagation delays in VLSI. IEEE J Solid-State Circuits SC-17:773–775 (1982)
18. Miller RE: Switching theory. Wiley, New York 1965
19. Miller RE: Chapter 10 [18], Vol. 2, pp 199–244
20. Molnar CE, Fang T-P: An asynchronous system design methodology. Technical Memorandum No 287. Computer Systems Laboratory, Washington University, St. Louis 1981
21. Molnar CE, Fang TP, Rosenberger FU: Synthesis of delay-insensitive modules. In: Fuchs H (ed) Proceedings 1985, Chapel Hill Conference on VLSI. Comput Sci Press 1985 pp 67–86
22. Peeters AMG: Decomposition of delay-insensitive circuits. Comput Sci Notes 90/04, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1990
23. Rem M: Concurrent computations and VLSI circuits. In: Broy M (ed) Control flow and data flow: concepts of distributed computing. Springer, Berlin Heidelberg New York 1985, pp 399–437
24. Rem M: Trace theory and systolic computations. In: de Bakker JW, Nijman AJ, Treleaven PC (eds) Proceedings PARLE, Parallel Architectures and Languages Europe, vol 1. Springer, Berlin Heidelberg New York 1987, pp 14–34
25. Schols HMJL: A formalisation of the foam rubber wrapper principle. Master's Thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1985
26. Seitz CL: Chapter 7, System Timing, in [16], pp 218–262
27. van de Snepscheut JLA: Trace theory and VLSI design. Lect Notes Comput Sci 200. Springer, Berlin Heidelberg New York 1985
28. Sutherland IE: Micropipelines. Commun ACM 32 (6):720–738 (1989)
29. Turing AM: Lecture to the London Mathematical Society on 20 February 1947. In: Carpentar BE, Doran RW (eds) Charles Babbage Institute Reprint Series for the History of Computing, vol 10. MIT Press, Cambridge, Massachusetts 1986
30. Udding JT: Classification and composition of delay-insensitive circuits. PhD Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands 1984
31. Udding JT: A formal model for defining and classifying delay-insensitive circuits and systems. Distrib Comput 1:197–204 (1986)
32. Unger SH: Asynchronous sequential switching circuits. Wiley, New York 1969
33. Verhoeff T: Characterizations of delay-insensitive communication protocols. Comput Sci Notes 89/6, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1989