EINDHOVEN
UNIVERSITY OF
TECHNOLOGY

The nature of delay-insensitive computing

Citation for published version (APA):
Rem, M. (1990). *The nature of delay-insensitive computing*. (Computing science notes; Vol. 9020). Technische Universiteit Eindhoven.

Document status and date:
Published: 01/01/1990

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: 16. Nov. 2023

The Nature of Delay-Insensitive Computing

by

Martin Rem

90/20

December, 1990

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science 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
editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.

The Nature of Delay-Insensitive Computing

Martin Rem
Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven
The Netherlands
e-mail: wsinrem@win.tue.nl

Abstract

Delay-insensitive systems are systems whose correct functioning does not depend on delay assumptions. In this paper a gradual introduction to delay-insensitivity is given, illustrated by many examples. Precise definitions are given of delay-insensitivity, decomposition (or refinement), and speed-independence. Recent results of the associated theory are touched upon.

1 Introduction

Almost all digital circuits contain clocks; not the types of clock that tell the time, but rather more like metronomes: in its simplest form a clock produces a periodic signal that alternates between a low and a high voltage level. Its high and low going transitions are used to synchronize different parts of the circuit.

Now imagine that the circuit has an input wire whose voltage level is sensed during the period when the clock is high, i.e. from a high going to the next low going transition. This sensing is done by producing the logical conjunction of the levels of the input wire and the clock. The result is stored in a flip-flop. A flip-flop is a device with two stable states; it enters one of these states depending on the level of the voltage it is offered.

If the input wire that is sensed happens to make a high going transition towards the end of the clock period, the voltage produced may be just a small 'runt' pulse, cf. Fig. 1. If the flip-flop is offered such a marginal pulse, it may linger for a while in a metastable state before entering one of its stable states. Unfortunately, there is no upper bound for the time the flip-flop may stay in the metastable state. This phenomenon is known as the *metastability phenomenon*[3,13]. It is sometimes referred to as the glitch phenomenon.

It is essential for clocked circuits that the clock period be chosen sufficiently long to guarantee that all parts of the circuit stabilize within the clock period. The metastability phenomenon obviously conflicts with this timing constraint.

The example above exhibits metastability in the presence of asynchronous inputs, but metastability also arises in arbitration and synchronization. An arbiter is a

device that is used to establish mutual exclusion among asynchronous requests. A synchronizer is a device that delays an asynchronous input in such a way that it is synchronized with another signal. The latter is usually the clock. Both arbiters and synchronizers can be realized only if we impose no upper bound on the time they take to produce their outputs. In essence, they do not produce their outputs until they have left the metastable states they possess.

In delay-insensitive systems we accept the fact that the durations of subcomputations may be unbounded. We, therefore, do not use an autonomous clock to synchronize the parts, but we have the different components of the system signal their completion explicitly[1]. We are aware that it may take quite some time before completions are signaled, but we cater to this by designing the system in such a way that its correct functioning does not depend on these delays.

A system consists of components and connecting wires. It is called *delay-insensitive* if it functions correctly under arbitrary and possibly varying delays in components and wires. Of course, the delays will affect the operating speed of the system, but this is not considered part of the 'correct functioning'. The type of correctness we do have in mind will be made precise in the sequel.

2 Communicating data

In order to acquire an operational appreciation of delay-insensitivity, we discuss the problem of delay-insensitively communicating data from one component to another. The problem is to send one bit of information from component S to component R, cf. Fig. 2.

As a first try, we connect the components by two wires: wire v to convey the bit, and wire r to signal that the data have been sent. The latter is known as a 'data valid' signal. Initially both wires are low. Component S first gives wire v the value of the bit to be communicated; after that it makes wire r high. Component R waits until wire r is high, after which it copies (for instance, into a flip-flop) the value of wire v.

The above scheme will solve the problem only if we know that the delay in wire v does not exceed that in wire r. Such a delay assumption, known as a 'bundling constraint' can, of course, not be made if we want the communication to be delay-

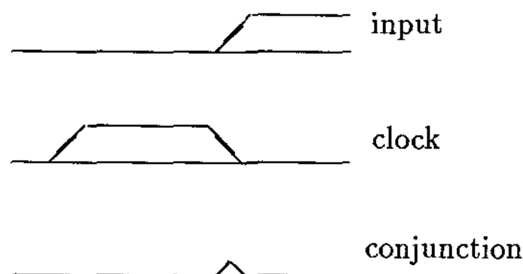

Figure 1: A 'runt' pulse

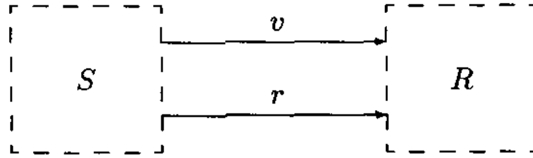

Figure 2: Communication with a data valid signal

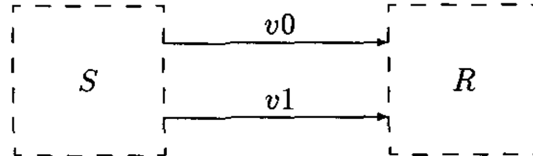

Figure 3: Dual rail communication

insensitive.

The solution is to code the bit to be communicated in such a way that R can detect its arrival[20]. This requires at least two wires to convey the bit: one wire can only have two states (low and high), but we need a third state to indicate the absence of a value. *Dual-rail encoding* is a technique that uses two wires per bit, cf. Fig. 3. The absence of a value is coded by two low wires. Value 0 is sent by making wire $v0$ high, and value 1 by making $v1$ high. The two wires are never high simultaneously.

The above scheme is not very useful if more bits have to be communicated successively: when may we decide that S can again send a bit? The only way out is to have R acknowledge that the bit has been received, cf. Fig. 4. Again, all wires are low initially. A complete cycle of sending one bit and acknowledging its receipt is now:

$$
\begin{array}{ll}
S: & vi\uparrow ; [a] ; vi\downarrow ; [\neg a] \\
R: & [v0 \vee v1] ; a\uparrow ; [\neg v0 \wedge \neg v1] ; a\downarrow
\end{array}
$$

Statement $vi\uparrow$ stands for 'make wire vi ($i = 0$ or $i = 1$) high' and, similarly, $vi\downarrow$ stands for 'make vi low'. Statement $[a]$ stands for 'wait until a holds', where high

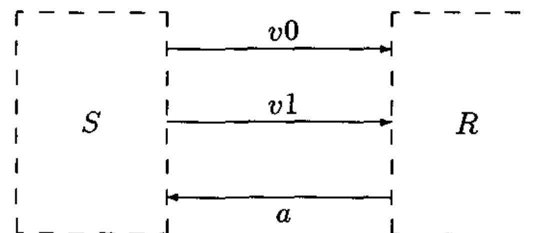

Figure 4: Communication with acknowledgement

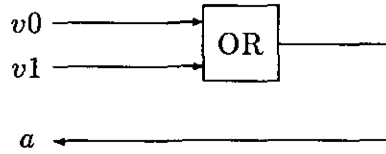

Figure 5: Generation of acknowledgement

and low are interpreted as *true* and *false*, respectively. In the above patterns we have not coded how S determines (at the beginning of its cycle) i, nor how R copies (at the first semicolon of its cycle) the value received. Notice that after a complete cycle all wires are low again. This form of signaling is known as *four-phase signaling.* Component R can generate signal a by using an OR-gate, cf. Fig. 5.

Component S initiates the communication by making wire vi high; S is the active partner in the communication. Component R starts with waiting for $v0$ or $v1$ to become high; this is the passive partner. In this case the distinction active/passive coincides with that of sender/receiver. This is not necessary: we can equally well have the sender be passive and the receiver active. A complete cycle then consists of

S : $[a]\,;vi\!\uparrow;[\neg a]\,;vi\!\downarrow$
R : $a\!\uparrow;[v0 \lor v1]\,;a\!\downarrow;[\neg v0 \land \neg v1]$

Now the receiver is the one that initiates the communication, viz. by making (request) wire a high. The sender does not start sending the bit until it has received this request. The schemes of active and passive sending are also known as data driven and demand driven, respectively.

3 C-element

The communication protocols developed above can easily be adapted for sending multiple-bit messages. We employ two wires per bit and extend the protocols straightforwardly, cf. Fig. 6. Since R acknowledges complete messages only, one acknowledge wire suffices.

We have seen that 1-bit messages can be acknowledged by means of an OR-gate. An interesting question is what mechanism we need for 2-bit messages. Consider the case that S is active. One may be tempted to generate signal a as the conjunction of $v0 \lor v1$ and $w0 \lor w1$, cf. Fig. 7.

This implementation, however, is erroneous. A possible sequence of events is

$$v0\!\uparrow;w0\!\uparrow;a\!\uparrow;v0\!\downarrow;a\!\downarrow$$

At this point the sender is allowed to transmit another message. However, the low going transition on $w0$ is still on its way, which can interfere with the next message. The problem is that the low going transition on a is generated too earley.

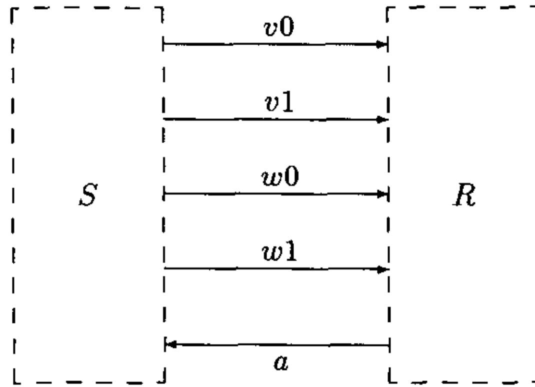

Figure 6: A 2-bit message

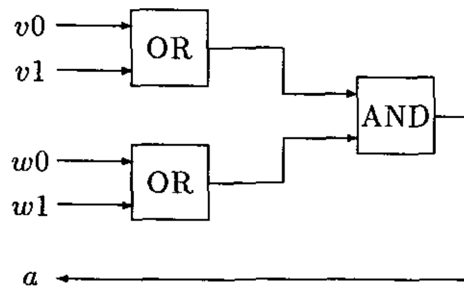

Figure 7: Erroneous implementation of acknowledgement

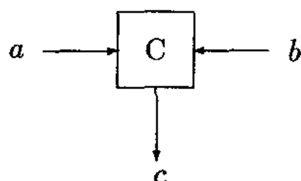

Figure 8: C-element

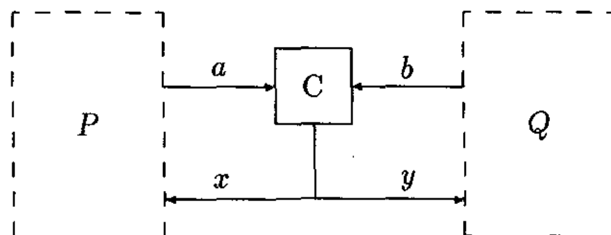

Figure 9: Synchronizing two components

Obviously, the AND-gate should be replaced by one that does not produce a low going transition on its outputs until both inputs have gone low.

Such an element is known as a Muller C-element, or simply C-element, cf. Fig. 8. It is sometimes called a last-of or a rendezvous element. If both inputs a and b have equal values, this value is also produced at output c; otherwise c remains what it was. This is a state-holding element: if the values at a and b differ, the value at c equals the last common value of the inputs.

A C-element is often used to synchronize different components, cf. Fig. 9. Components P and Q have to be synchronized to accomplish 'mutual inclusion', i.e., they each have a synchronization point at which they must wait for the other component to reach its synchronization point. This can be realized by the following protocol for P:

$$a\uparrow\,;[x]\,;S\,;a\downarrow\,;[\neg x]$$

and similarly for Q. Statement S represents the part that is executed in mutual inclusion with component Q.

4 Think transitions

Above we have tried to give a conventional description of a C-element, viz. by giving how the output values depend on the input values. Such descriptions, however, are not very adequate for use in delay-insensitive systems. In delay-insensitive systems the transitions are the important events, and what should be specified are the possible orders in which these events may take place[15]. For the C-element these possible orders may be specified by the following behavioral expression:

$$(a\uparrow, b\uparrow\,; c\uparrow\,; a\downarrow, b\downarrow\,; c\downarrow)^*$$

It expresses that first input wires a and b go high (the comma, which takes priority over the semicolon, expresses concurrency), after which output wire c goes high (the semicolon expresses order), which is followed by a and b going low, after which c goes low. From then on it starts all over again (the asterisk expresses repetition). The assumption is again that initially all wires are low. If we neglect the directions of the transitions the above expression may be written as

$$(a, b\,; c)^*$$

We draw a scheme that shows how the values on the output wires depend on those on the input wires, writing 'low' as 0 and 'high' as 1:

a	b	c
0	0	0
1	0	0
0	1	0
1	1	1
0	1	1
1	0	1

The fact that we have different output values for the same input combination shows that C-elements are indeed sequential (or state-holding) elements.

A behavioral expression specifies an interface between a component and its environment. It specifies when the component may produce output transitions, but it also specifies when its environment may offer input transitions: input transitions are not allowed to arrive at 'wrong moments'. If an input transition arrives 'out of order' this is called *computation interference.* Now it is becoming clear what we mean by 'correct functioning' of a system. A system consists of components, each specified by the possible orders in which the transitions may occur. The components should be such that the system cannot exhibit computation interference.

In delay-insensitive systems one usually discerns a second correctness requirement, besides absence of computation interference, and that is absence of transmission interference. We speak of *transmission interference* if there is a connecting wire at which there are at least two transitions simultaneously present. We can phrase transmission interference as a form of computation interference by saying that each wire from point a to point b is a component with

$$(a\uparrow\,; b\uparrow\,; a\downarrow\,; b\downarrow)^*$$

or simply $(a\,; b)^*$, as its behavioral expression.

The behavioral expression does not give a complete description of what a component 'can do'. Consider, for example, the following expression:

$$(a?\,; c!\,; b?\,; d!)^*$$

Symbols '?' and '!' specify that a and c are inputs and b and d outputs. We have not mentioned the directions of the transitions. This component can be implemented by just two wires that connect a with c and b with d. The same two wires would, however, also implement, for example,

$$(a?\,;c!\,;a?\,;c! \;\mid\; b?\,;d!\,;b?\,;d!)^*$$

where the bar denotes the choice-operator, similar to the plus in regular expressions. The bar has a lower priority than the comma and the semicolon.

Next replace in the above expression d by c, so that only one output remains:

$$(a?\,;c!\,;a?\,;c! \;\mid\; b?\,;c!\,;b?\,;c!)^*$$

This component may be implemented by an OR-gate, as the following table shows:

a	b	c
0	0	0
1	0	1
0	1	1

In contrast to that of the C-element, this table exhibits exactly one output value per input combination. Such processes are called *combinational.*

5 Formal definition of processes and systems

Before giving a formal (operational) definition of delay-insensitivity, we must first define what processes and systems are. We use a simple trace-theoretic model for processes:

A *process* T, sometimes referred to as a directed process, is a triple $\langle I, O, T\rangle$ such that

$$I \cap O = \emptyset$$
$$T \subseteq (I \cup O)^*$$
$$T \neq \emptyset$$
$$T \text{ prefix-closed}$$

Set I is the set of input symbols and O the set of output symbols. The elements of T are finite-length sequences, known as *traces*, of elements in $I \cup O$. Trace set T is called *prefix-closed* if $sa \in T \Rightarrow s \in T$ for $a \in I \cup O$.

Example 1 Consider process $\langle I, O, T\rangle$ with

$$I = \{a, b\}$$
$$O = \{c\}$$
$$T = \{\varepsilon, a, b, ab, ba, abc, bac, abca, baca, \cdots\}$$

where ε denotes the empty trace. This process is a C-element. We usually specify it by the behavioral expression

$$(a?, b?\,;c!)^*$$

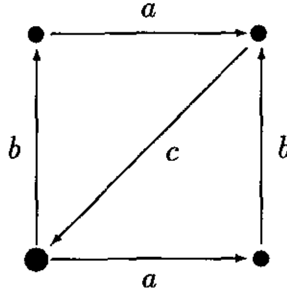

Figure 10: State graph of a C-element

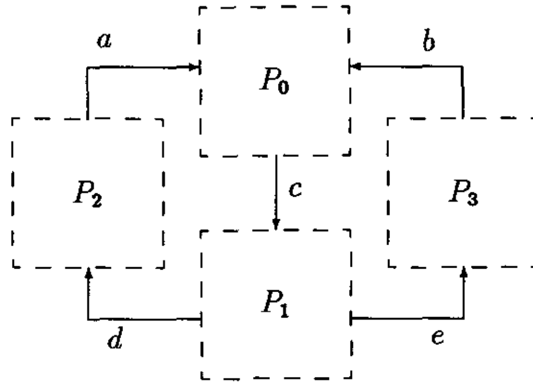

Figure 11: A system of four processes

Its trace set consists of all sequences of symbols one encounters when traversing the graph of Fig. 10, starting in the lower left-hand corner.

A *system* is a set of processes, such that each symbol of a process occurs in exactly one process as input symbol and in exactly one process as output symbol. The connecting wires are not modeled explicitly; each symbol represents a wire, running from the process of which it is an output symbol to the process of which it is an input symbol. Thus we have defined what is known as a closed system (no dangling inputs or outputs) with point-to-point connections. Both conditions may be weakened, but the restricted definition suffices for our purposes.

Example 2 Consider the system consisting of four processes specified by

P_0 : $(a?, b? \,; c!)^*$
P_1 : $(d! \,; e! \,; c?)^*$
P_2 : $(d? \,; a!)^*$
P_3 : $(e? \,; b!)^*$

Process P_0 is a C-element. A pictorial impression of the system is shown in Fig. 11.

Definition of delay-insensitivity Consider a system of n processes: $P_0, P_1, \cdots, P_{n-1}$, where $P_i = \langle I_i, O_i, T_i \rangle$. The *states* of the system are the n-tuples $\langle t_0, t_1, \cdots, t_{n-1} \rangle$

with $t_i \in (I_i \cup O_i)^*$. We define the *reachable states* of the system as follows:

1) $\langle \varepsilon, \varepsilon, \cdots, \varepsilon \rangle$ is reachable

2) if $\langle t_0, \cdots, t_i, \cdots, t_{n-1} \rangle$ is reachable $(0 \leq i < n)$ and
$a \in O_i \ \wedge \ t_i a \in T_i$
or
$a \in I_i \cap O_j \ \wedge \ a \# t_j > a \# t_i$
then $\langle t_0, \cdots, t_i a, \cdots, t_{n-1} \rangle$ is reachable

3) no other states are reachable

where $a\#t$ denotes the number of occurrences of symbol a in trace t.

The idea behind the above definition is that in state $\langle t_0, t_1, \cdots, t_{n-1} \rangle$ trace t_i is the current trace of process P_i. Condition 1) expresses that the initial state is reachable. In the course of a computation current traces are extended only. They can be extended with output symbols and with input symbols. The rule governing these extensions distinguishes output and input. Condition 2) expresses that the current trace of a process may be extended with an output symbol if the extended trace belongs to the trace set of the process. Notice that the prefix-closedness implies that then the current trace was in the trace set as well. The second part of 2) expresses that the current trace may be extended with an input symbol if that symbol happens to be 'on its way', i.e. if it has been output more often than it has been received. This extension may lead to a current trace that is not in the trace set of the process. The reception of an input is actually the only way to bring the current trace outside the trace set. The model captures that processes do control (by their trace sets) the sending of outputs but not the reception of inputs.

Examples of reachable states for the system of Example 2 are

$$\langle \varepsilon, \varepsilon, \varepsilon, \varepsilon \rangle$$
$$\langle \varepsilon, d, \varepsilon, \varepsilon \rangle$$
$$\langle \varepsilon, de, \varepsilon, \varepsilon \rangle$$
$$\langle \varepsilon, de, \varepsilon, e \rangle$$
$$\langle \varepsilon, de, \varepsilon, eb \rangle$$
$$\langle b, de, \varepsilon, eb \rangle$$

We have now all ingredients to define delay-insensitivity for systems. State $\langle t_0, t_1, \cdots, t_{n-1} \rangle$ is called *safe* if

$$(\forall j : 0 \leq j < n : t_j \in T_j)$$
$$\wedge$$
$$(\forall a, i, j : a \in I_i \cap O_j : a\#t_j \leq a\#t_i + 1)$$

The first condition expresses the absence of computation interference and the second one the absence of transmission interference. A system is called *delay-insensitive* if all its reachable states are safe.

The system of Example 2 is an example of a delay-insensitive system. The following example is not delay-insensitive. Process $\tilde{P}$ denotes the *reflection* of process P, i.e. if $P = \langle I, O, T \rangle$ then $\tilde{P} = \langle O, I, T \rangle$.

Example 3 Consider the system consisting of process P_1 of Example 2 and its reflection:

P_1 : $(d!\,;e!\,;c?)^*$
$\tilde{P}_1$: $(d?\,;e?\,;c!)^*$

Reachable states are

$$\langle\varepsilon,\varepsilon\rangle$$
$$\langle d,\varepsilon\rangle$$
$$\langle de,\varepsilon\rangle$$
$$\langle de,e\rangle$$

However, the latter state is not safe; computation interference has occurred: e is not a trace of proces $\tilde{P}_1$. The system is, consequently, not delay-insensitive.

Example 4 An example of a system with transmission interference is $\{P_0, P_1\}$:

P_0 : $(a!, b?)^*$
P_1 : $(a?, b!)^*$

The following table shows some reachable states of this system:

P_0	a		b	a	...
P_1		b			

The vertical lines correspond to reachable states, viz. from left to right: $\langle\varepsilon,\varepsilon\rangle$, $\langle a,\varepsilon\rangle$, $\langle a,b\rangle$, $\langle ab,b\rangle$, $\langle aba,b\rangle$, i.e. time goes to the right and the rows of symbols represent current traces of the processes listed in the first column. Since $a\#aba > a\#b+1$, the latter state exhibits transmission interference.

6 Decomposition

Suppose a computation is specified as a process and we have to design a delay-insensitive implementation for it. In other words, we have to find a set of processes into which the specified process can be decomposed [21,12,11,8,18].

Let P be a process and let X be a set of processes such that $\tilde{P} \notin X$. We define set X to be a *decomposition* of process P if set $X\cup\{\tilde{P}\}$ is a delay-insensitive system.

Example 5 As a first example of a decomposition we consider set $\{P_0, P_1\}$:

P_0 : $(a?\,;b!)^*$
P_1 : $(b?\,;c!)^*$

This is a decomposition of

Q : $(a?\,;c!)^*$

Consider the system consisting of processes P_0, P_1, and $\tilde{Q}$. Its reachable states are given by the following table:

P_0		a	b					a	
P_1				b	c				$\cdots$
$\tilde{Q}$	a	a				c	a		

where $\tilde{Q}$ is the process given by $(a!\,;c?)^*$. All reachable states are safe. The example shows that a wire may be decomposed into two connected wires.

Example 6 Next we consider two unconnected wires. Let processes P_0 and P_1 be given by

P_0 : $(a?\,;c!)^*$
P_1 : $(b?\,;d!)^*$

Set $\{P_0, P_1\}$ is a decomposition of

Q : $(a?\,;c!\,;b?\,;d!)^*$

as the following table of reachable states shows:

P_0		a	c							
P_1						b	d			$\cdots$
$\tilde{Q}$	a			c	b			d	a	

where $\tilde{Q}$ is the process given by $(a!\,;c?\,;b!\,;d?)^*$. It is, however, also a decomposition of, for example,

$$(a?\,;c!\,;a?\,;c! \;\mid\; b?\,;d!\,;b?\,;d!)^*$$

as can be easily checked. This proves the claim made in Section 4. It also shows that composition cannot simply be the inverse of decomposition. A suitable definition of composition can be found in [17,4].

Example 7 A 3-input C-element can be decomposed into two 2-input C-elements:

Q : $(a?, b?, c?\,;e!)^*$
P_0 : $(a?, b?\,;d!)^*$
P_1 : $(c?, d?\,;e!)^*$

Now Q decomposes into $\{P_0, P_1\}$, as can be checked easily.

A decomposition rule is useful only if it satisfies the *substitution property.* This property states that if process P decomposes into $X \cup \{Q\}$ and process Q decomposes into Y then P decomposes into $X \cup Y$. Our decomposition rule indeed satisfies the substitution property, provided that distinct names are used for the internal wires in X and Y.

Example 8 In this example a process is decomposed into a set of just one process. In other words, the latter process implements, or 'refines', the other process.

Consider process P, given by

$$P: \qquad (a?\,;(b! \mid c!))^*$$

and process $Q = \langle I, O, T \rangle$ with $I = \{a\}$, $O = \{b, c\}$, and T given by

$$(a?\,;b!)^*$$

Process Q differs from process P in that it does not produce output c. Process P can be decomposed into process Q, as the following table shows:

P		a	b			...
$\tilde{Q}$	a			b	a	

This example demonstrates that in the choice between outputs the designer is allowed to make an a priori choice. The word 'allowed' means here, of course: without running the risk of causing computation or transmission interference, since these are the only correctness concerns we have introduced. In particular have we not considered progress requirements.

A designer is not allowed to make an a priori choice between inputs. For example, process $\tilde{P}$ does not decompose into $\tilde{Q}$:

$\tilde{Q}$	a			c	...
P		a	c		

Here we have computation interference: ac is not a trace of $\tilde{Q}$. As an aside we mention that $\tilde{Q}$ does decompose into $\tilde{P}$.

An interesting question is whether a process decomposes into itself. This is in general not the case. Process P_1 of Example 2 is a process that does not decompose into itself, as we observed in Example 4.

Processes that decompose into themselves are known as *delay-insensitive* processes. The C-element is an example of a delay-insensitive process. There are several characterizations of delay-insensitive processes, the oldest of which was given by J.T. Udding[16]. As we have seen in Example 2, processes that are not delay-insensitive can very well be used to construct delay-insensitive systems.

7 Building blocks

The typical way of designing an inverter in CMOS is shown in Fig. 12. The input is forked to two transistors. This is clearly not a delay-insensitive decomposition of an inverter into two transistors: if one of the two branches of the fork is exceptionally slow a conveying connection between power and ground is maintained, a situation that is more commonly known as a short circuit.

Individual transistors are simply too primitive to be used as building blocks for delay-insensitive compositions. Delay-insensitive systems require building blocks of a higher aggregation level. Ebergen[5] has outlined a finite set of building blocks

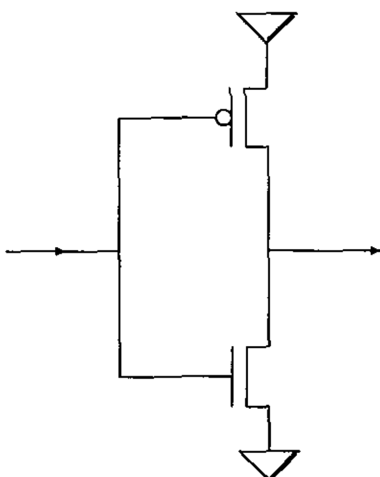

Figure 12: A CMOS inverter

into which all delay-insensitive processes can be decomposed. This set consists of two types of C-elements, a fork, an exclusive OR, a toggle, and an arbiter. Internally such building blocks will not be delay-insensitive. They correspond to what Seitz[14] has termed *equipotential regions.*

As mentioned in Section 4, combinational processes are processes that have exactly one output combination for each combination of input values. An example of a combinational process is

$M: \qquad (a?, b?\,; d!\,; c?\,; e!)^*$

as the following table of input values and corresponding output values shows:

a	b	c	d	e
0	0	0	0	0
1	0	0	0	0
0	1	0	0	0
1	1	0	1	0
1	1	1	1	1
0	1	1	1	1
1	0	1	1	1
0	0	1	0	1

M is a process with two outputs. According to the table above, output d is the majority of the input values and output e is a copy of input c. Let process P be specified by $(d?\,; c!)^*$. Then C-element $(a?, b?\,; e!)^*$ can be decomposed into $\{M, P\}$:

M			a	b	d			c	e				
P						d	c						$\cdots$
$\tilde{C}$	a	b								e	a	b	

Thus we have exhibited a delay-insensitive decomposition of a sequential process into two combinational processes.

Brzozowski and Ebergen[2] have shown that sequential processes cannot be decomposed into sets that contain only forks, i.e. processes of the form $(a?\,;b!,c!)^*$, and single-output combinational processes. Martin[9] shows that extending these sets with C-elements does not help very much. Essentially, the only sequential processes that can then be built are various forms of C-elements.

8 Speed-independent

In the speed-independent computing model, which is older than the delay-insensitive one[10], all delays are assumed to be in the components. The wires do not exhibit delay, which makes transmission interference not an issue.

In order to define speed-independence more precisely, we need to change our definition of reachable states (which models asynchronous communication) into one that is based on synchronous communication. For *synchronously reachable* the second condition in the definition of reachable reads:

2) if $\langle t_0, \cdots, t_i, \cdots, t_j, \cdots, t_{n-1}\rangle$ is reachable and
$a \in O_i \cap I_j \ \wedge\ t_i a \in T_i$
then $\langle t_0, \cdots, t_i a, \cdots, t_j a, \cdots, t_{n-1}\rangle$ is reachable

A state $\langle t_0, t_1, \cdots, t_{n-1}\rangle$ is called *safe* if

$$(\forall j : 0 \leq j < n : t_j \in T_j)$$

A system is called *speed-independent* if all states that are synchronously reachable are safe.

The reachable states under synchronous communication form a subset of those that are reachable under asynchronous communication. Delay-insensitive systems are, consequently, also speed-independent. The inverse is not true.

We show that a C-element can speed-independently be decomposed into a single-output combinational process P_0 and a fork P_1[6]:

P_0 : $(a?, b?\,; d!\,; e?)^*$
P_1 : $(d?\,; e!\,; c!)^*$

Process P_0 is combinational, as the following table shows:

a	b	e	d
0	0	0	0
1	0	0	0
0	1	0	0
1	1	0	1
1	1	1	1
1	0	1	1
0	1	1	1
0	0	1	0

Process P_1 is a kind of fork that is (in speed-independent settings) often referred to as an *isochronic fork*. In order to demonstrate that C-element

C : $(a?, b? \,; c!)^*$

can speed-independently be decomposed into $\{P_0, P_1\}$, we investigate system $\{P_0, P_1, \tilde{C}\}$, with $\tilde{C}$ given by $(a!, b! \,; c?)^*$. This system is indeed speed-independent:

P_0	a	b	d	e		a	
P_1			d	e	c		...
$\tilde{C}$	a	b			c	a	

System $\{P_0, P_1, \tilde{C}\}$ is not delay-insensitive. An important difference between speed-independence and delay-insensitivity is that in the speed-independent model we can realize forks that guarantee that one of its outputs arrives earlier at a component than the other one does.

9 Conclusion

Starting with the problem of communicating data, we have gradually found our way to an operational, but precise, definition of delay-insensitivity. The virtue of this operational model is not only its relative simplicity, but also its clear relation with computing media in general and VLSI circuitry in particular. We have used trace theory[19,7] to formulate these definitions, since traces are very well-suited to express nontemporal relations between events. Our treatment exhibits a clear separation between the *communication model*, which captures the types of delays we want the correctness of the system to be independent of, and the *correctness concerns*. We have discussed two communication models: one in which the delays are both in the components and in the wires, and one in which the delays are just in the wires. With respect to correctness we have, throughout the paper, sticked to just one correctness concern: absence of interference.

Design is nothing else than decomposing large problems into smaller ones, until the latter problems either are trivial or have been solved before. Therefore, we have extensively addressed the concept of decomposition, interleaved with many examples. There is a limit to delay-insensitivity: one ends up with primitive building blocks of one kind or another. We have briefly discussed the nature of these blocks.

10 Acknowledgements

I am indebted to Tom Verhoeff, who is the inspirator behind the operational model in this paper. Ivan Sutherland coined the title of Section 4. Acknowledgements are also due to Kees van Berkel and the members of the Eindhoven VLSI Club for numerous discussions on the ins and outs of delay-insensitivity.

References

[1] Clifford Barney. Logic designers toss out the clock. *Electronics*, Dec. 9, 1985, 42–45

[2] J.A. Brzozowski and J.C. Ebergen. *On the Delay-Sensitivity of Gate Networks.* Computing Science Note 90/5, TU Eindhoven, 1990

[3] T.J. Chaney and C.E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, Vol. C-22, 1973, 421–422

[4] W. Chen, J.T. Udding, and T. Verhoeff. Networks of communicating processes and their (de)-composition in *The Mathematics of Program Construction* (J.L.A. van de Snepscheut, ed.). LNCS 375, Springer-Verlag, 1989, 174–176

[5] J.C. Ebergen. *Translating Programs into Delay-Insensitive Circuits.* CWI Tract 56, CWI, Amsterdam, 1989

[6] Mark B. Josephs. *Receptive Process Theory.* Computing Science Note 90/8, TU Eindhoven, 1990

[7] Anne Kaldewaij. *A Formalism for Concurrent Processes.* Ph.D. Thesis, TU Eindhoven, 1986

[8] Alain J. Martin. Compiling communicating processes into delay-insensitive circuits. *Distributed Computing,* **1**, 1986, 247–260

[9] Alain J. Martin. The limitations of delay-insensitivity in asynchronous circuits in *Beauty Is Our Business* (W.H.J. Feijen et al., eds.) Springer-Verlag, 1990, 302–311

[10] R.E. Miller. *Switching Theory,* Vol. 2, Wiley, 1965

[11] Charles E. Molnar, Ting-Pien Fang and Frederick U. Rosenberger. Synthesis of delay-insensitive modules in *1985 Chapel Hill Conference on Very Large Scale Integration* (Henry Fuchs, ed.) Computer Science Press, 1985, 67–86

[12] Martin Rem. Concurrent computations and VLSI circuits in *Control Flow and Data Flow* (M. Broy, ed.) Springer-Verlag, 1985, 399–437

[13] Science and the citizen. *Scientific American,* **228**, April 1973, 43–44

[14] C.L. Seitz. System timing in Carver Mead and Lynn Conway, *Introduction to VLSI Systems.* Addison-Wesley, 1980, 218–262

[15] I.E. Sutherland. Micropipelines. *Commun. ACM,* **32**, 1989, 720–738

[16] Jan Tijmen Udding. A formal model for defining and classifying delay-insensitive circuits and systems. *Distributed Computing,* **1**, 1986, 197–204

[17] Jan Tijmen Udding and Tom Verhoeff. *The Mathematics of Directed Specifications.* Technical Report WUCS 88-20, Washington University, 1988

[18] C.H. (Kees) van Berkel and Ronald W.J.J. Saeijs. Compilation of communicating processes into delay-insensitive circuits in *1988 IEEE Int. Conf. on Computer Design,* IEEE Computer Society Press, 1988, 157–162

[19] Jan L.A. van de Snepscheut. *Trace Theory and VLSI Design.* LNCS 200, Springer-Verlag, 1985

[20] Tom Verhoeff. Delay-insensitive codes—an overview. *Distributed Computing*, **3**, 1988, 1–8

[21] Alexandre Yakovlev. Designing self-timed systems. *VLSI Systems Design*, September 1985, 70–90

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.L.J.Schols	Delay insensitive directed trace structures satisfy the foam 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 specification 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 systems.
86/05	J.L.G. Dietz K.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

86/14	R. Koymans	Specifying passing systems requires extending temporal logic.
87/01	R. Gerth	On the existence of sound and complete axiomati zations 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 distributed 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 nondeterministic 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.
88/11	G.J. Houben J.Paredaens	A graphical interface formalism: specifying nested relational databases.
88/12	A.E. Eiben	Abstract theory of planning.
88/13	A. Bijlsma	A unified approach to sequences, bags, and trees.
88/14	H.M.M. ten Eikelder R.H. Mak	Language theory of a lambda-calculus with recursive types.

88/15	R. Bos C. Hemerik	An introduction to the category theoretic solution of recursive domain equations.
88/16	C.Hemerik J.P.Katoen	Bottom-up tree acceptors.
88/17	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable specifications for discrete event systems.
88/18	K.M. van Hee P.M.P. Rambags	Discrete event systems: concepts and basic results.
88/19	D.K. Hammer K.M. van Hee	Fasering en documentatie in software engineering.
88/20	K.M. van Hee L. Somers M.Voorhoeve	EXSPECT, the functional part.
89/1	E.Zs.Lepoeter-Molnar	Reconstruction of a 3-D surface from its normal vectors.
89/2	R.H. Mak P.Struik	A systolic design for dynamic programming.
89/3	H.M.M. Ten Eikelder C. Hemerik	Some category theoretical properties related to a model for a polymorphic lambda-calculus.
89/4	J.Zwiers W.P. de Roever	Compositionality and modularity in process specification and design: A trace-state based approach.
89/5	Wei Chen T.Verhoeff J.T.Udding	Networks of Communicating Processes and their (De-)Composition.
89/6	T.Verhoeff	Characterizations of Delay-Insensitive Communication Protocols.
89/7	P.Struik	A systematic design of a paralell program for Dirichlet convolution.
89/8	E.H.L.Aarts A.E.Eiben K.M. van Hee	A general theory of genetic algorithms.
89/9	K.M. van Hee P.M.P. Rambags	Discrete event systems: Dynamic versus static topology.
89/10	S.Ramesh	A new efficient implementation of CSP with output guards.
89/11	S.Ramesh	Algebraic specification and implementation of infinite processes.
89/12	A.T.M.Aerts K.M. van Hee	A concise formal framework for data modeling.

89/13	A.T.M.Aerts K.M. van Hee M.W.H. Hesen	A program generator for simulated annealing problems.
89/14	H.C.Haesen	ELDA, data manipulatie taal.
89/15	J.S.C.P. van der Woude	Optimal segmentations.
89/16	A.T.M.Aerts K.M. van Hee	Towards a framework for comparing data models.
89/17	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra.
90/1	W.P.de Roever-H.Barringer C.Courcoubetis-D.Gabbay R.Gerth-B.Jonsson-A.Pnueli M.Reed-J.Sifakis-J.Vytopil P.Wolper	Formal methods and tools for the development of distributed and real time systems, pp. 17.
90/2	K.M. van Hee P.M.P. Rambags	Dynamic process creation in high-level Petri nets, pp. 19.
90/3	R. Gerth	Foundations of Compositional Program Refinement - safety properties - , p. 38.
90/4	A. Peeters	Decomposition of delay-insensitive circuits, p. 25.
90/5	J.A. Brzozowski J.C. Ebergen	On the delay-sensitivity of gate networks, p. 23.
90/6	A.J.J.M. Marcelis	Typed inference systems : a reference document, p. 17.
90/7	A.J.J.M. Marcelis	A logic for one-pass, one-attributed grammars, p. 14.
90/8	M.B. Josephs	Receptive Process Theory, p. 16.
90/9	A.T.M. Aerts P.M.E. De Bra K.M. van Hee	Combining the functional and the relational model, p. 15.
90/10	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).
90/11	P. America F.S. de Boer	A proof system for process creation, p. 84.
90/12	P.America F.S. de Boer	A proof theory for a sequential version of POOL, p. 110.
90/13	K.R. Apt F.S. de Boer E.R. Olderog	Proving termination of Parallel Programs, p. 7.
90/14	F.S. de Boer	A proof system for the language POOL, p. 70.
90/15	F.S. de Boer	Compositionality in the temporal logic of concurrent systems, p. 17.

90/16	F.S. de Boer C. Palamidessi	A fully abstract model for concurrent logic languages, p. 23.
90/17	F.S. de Boer C. Palamidessi	On the asynchronous nature of communication in concurrent logic languages: a fully abstract model based on sequences, p. 29.
90/18	J.Coenen E.v.d.Sluis E.v.d.Velden	Design and implementation aspects of remote procedure calls, p. 15.
90/19	M.M. de Brouwer P.A.C. Verkoulen	Two Case Studies in ExSpect, p. 24.
90/20	M.Rem	The Nature of Delay-Insensitive Computing, p.18.
90/21	K.M. van Hee P.A.C. Verkoulen	Data, Process and Behaviour Modelling in an integrated specification framework, p.