Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets

Citation for published version (APA):
Eindhoven University of Technology
Department of Mathematics and Computing Science

Derivation of delay insensitive and speed independent CMOS circuits,
using directed commands and production rule sets

by

Wilbert Körver 93/31

Computing Science Note 93/31
Eindhoven, September 1993
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.

Copies can be ordered from:
Mrs. M. Philips
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

All rights reserved
editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.
## CONTENTS

Abstract 1

1 Design Theory 1
   1.1 Delay Insensitivity and Speed Independence 2
   1.2 Decompositions and DI-decompositions 3
       Directed Trace Structures and Directed Commands 3
       Specifications and States 4
       Conditional Transitions 5
       A Heuristic for finding Decompositions 6
   1.3 Production Rule Sets to derive (C) MOS Implementations 7
       Boolean Terms and Literal Terms 7
       Production Rules 8
       Translation of P.R. Sets into (C) MOS Circuits 10
       Calculating with P.R. Sets 11
   1.4 Costs Measures for (C) MOS Implementations 16
       Size Measure 16
       Delay Measure 18
       Examples 19

2 The Demultiplexer 20
   2.1 Introduction 20
   2.2 Derivations 21
       2.2.1 Solution 1 22
           Derivation 22
           Generalization 23
       2.2.2 Solution 2 24
           Derivation 24
           Global isochronic forks 25
           Generalization 26
       2.2.3 Solution 3 27
           Derivation 27
           Implementations 29
           Global isochronic forks 32
           Generalization 33
   2.3 Comparison 34

3 Concluding Remarks 35

Appendix A Proof of theorem 1 36

Acknowledgements 39

References 39
Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets

Wilbert H.F.J. Körver
Department of Computer Science
Leiden University
Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
e-mail: korver@rulwinw.LeidenUniv.nl
September 1993

ABSTRACT
The subject of this paper is the design and implementation of delay insensitive and speed independent circuits. A circuit is specified by a directed command satisfying certain conditions. Ebergen's decomposition method ([JE1,2,3]) is used to find globally delay insensitive and locally speed independent networks of components implementing the circuit. An extension of the production rules from [AM1,2] is used to derive speed independent (C)MOS implementations (in terms of transistor networks) for components. Finally, cost measures are given to compare the different (C)MOS implementations. Using these methods a number of implementations of the demultiplexer (a basic component in [B&S]) are derived and compared.

1 Design Theory
This paper presents a design method for CMOS transistor networks. Starting with abstract specifications in terms of directed commands, the first steps consist of the design of globally delay insensitive and locally speed independent circuits. For these steps, the decomposition method from [JE1,2,3] is used. This method is extended with conditional transitions and some notions on states, which are directed towards the state-based arguments that are convenient at lower levels (transistor networks) rather than towards the transition-based arguments used at higher levels (networks of components).

The last step is the derivation of transistor networks from directed commands. For this step, production rule sets are presented, which can be derived from directed commands, and can be translated into CMOS transistor networks. Some calculation rules for production rule sets are given and discussed. Finally, abstract size and delay measures for transistor networks are given.

---

1 A draft version of this paper appeared in September 1990 with as title 'Derivation of Implementations of the Demultiplexer'. The author previously worked at the Eindhoven University, where this research was done.
The paper is organized as follows. In section 1.1 delay insensitivity and speed independence are explained and a general design strategy is given. In the remainder of chapter 1, the theory described above is introduced. In chapter 2 the theory is illustrated by means of a 'larger' example, where the main emphasis is on demonstrating the application of the theory, and not so much on the formal correctness proofs related to this application.

1.1 Delay Insensitivity and Speed Independence

A circuit is formally regarded as a network of components. A circuit is called delay insensitive if the correctness of functional operation of the circuit is insensitive to delays in both wires and components. A circuit is called speed independent if the correctness of functional operation of the circuit is insensitive to delays in components. In the formalization of speed independent circuits the delays in wires are modelled to be zero, i.e., the communication between components is assumed to be instantaneous.

In practice, however, wires in speed independent circuits do have delays. Speed independent circuits must, therefore, be implemented such that (A) the delays in wires do not change the (relative time-) order of signals, and (B) the delays in the different branches of a fork are such that the receipt of a transition at all connected components is completed before one of the components produces a resulting output.

Forks that behave as described in (B) are called isochronic forks (they are depicted as normal forks labelled with an equality sign). In our derivations we will avoid the case that the order of signals can be changed merely by wire delays, and, hence, case (A) need not be taken into consideration. As a result, the main difference in our designs between speed independent circuits and delay insensitive circuits is the occurrence of isochronic forks.

A formal circuit (a network of components) is an abstraction of a physical circuit. In the derivation of an implementation (a physical circuit), subcomponents are decomposed, thus deriving another formal circuit. This way, a number of hierarchically ordered abstractions of the same physical circuit are considered. At different levels of abstraction, the circuits we design are delay insensitive (at higher levels) or speed independent (at lower levels).

The, from an implementational point of view, most important subcomponents of the circuits we design are called cells. A cell must be implemented at a small and local area of the physical circuit. That way, the wire delay constraints of speed independent circuits (in forks) can be handled. Hence, a cell may be either delay insensitive or speed independent, but it must be relatively small in order to be implementable as a speed independent circuit.

We do not use any constraints on the relative location of distinct cells. Consequently, assumptions on the delays in wires connecting cells are not allowed. Therefore, in order for the design to be implementable, the communication between cells must be delay insensitive.
In other words, the circuit, viewed as a connection of cells, must be delay insensitive.

An isochronic fork between cells, i.e. an isochronic fork with different outgoing branches leading to distinct cells, is called a \textit{global isochronic fork}. An isochronic fork within a cell is called a \textit{local isochronic fork}. As will be clear from the above, our design strategy allows local isochronic forks and avoids global isochronic forks. The implementation of isochronic forks requires careful consideration, cf. \cite{CvB2}.

1.2 Decompositions and DI-decompositions

To derive speed independent and delay insensitive circuits we use Ebergen's decomposition methods (see \cite{JE1,2,3}). Two main notions in his theory are \textit{decomposition} and \textit{DI-decomposition}. A decomposition corresponds to a realization of a component by means of a speed independent network of components. A DI-decomposition corresponds to a realization of a component by means of a delay insensitive network of components. For an introduction to Ebergen's decomposition method we refer to \cite{JE1,2,3}.

In this section we briefly explain the formalism underlying his theory (partly from \cite{JE2}). We introduce a new notion, called 'conditional transition', and some notions on specifications and states. We summarize the heuristic for finding decompositions given in \cite{JE3}, and make some remarks on this heuristic.

Directed Trace Structures and Directed Commands

A \textit{directed trace structure} is a triple \((A, B, X)\), where \(A\) and \(B\) are finite sets of symbols and \(X \subseteq (A \cup B)^*\). The set \((A \cup B)^*\) is the set of all finite traces of symbols from \(A \cup B\). The empty trace is denoted by \(\varepsilon\). For a directed trace structure \(R = (A, B, X)\), the set \(A \cup B\) is called the \textit{alphabet} of \(R\) and denoted by \(aR\); the set \(A\) is called the \textit{input alphabet} of \(R\) and denoted by \(iR\); the set \(B\) is called the \textit{output alphabet} of \(R\) and denoted by \(oR\); the set \(X\) is called the \textit{trace set} of \(R\) and denoted by \(tR\).

In order to construct more complex directed trace structures from simpler ones we use the following operations:

- **Concatenation**: \(R; S = \langle iR \cup iS, oR \cup oS, \{t_0t_1 \mid t_0 \in tR \land t_1 \in tS\} \rangle\)
- **Union**: \(R \cup S = \langle iR \cup iS, oR \cup oS, tR \cup tS \rangle\)
- **Repetition**: \([R] = \langle iR, oR, (tR)^* \rangle\)
- **Prefix-closure**: \(\text{pref} R = \langle iR, oR, \{t_0 \mid (E \ t_1:: t_0 \in tR)\} \rangle\)
- **Projection**: \(R \mid A = \langle iR \mid A, oR \mid A, \{t \mid t \in tR\} \rangle\)
- **Weaving**: \(R \| S = \langle iR \cup iS, oR \cup oS, \{t(e(aR \cup aS)^* \mid t \in tR \land t \in tS)\} \rangle\)

where \(R\) and \(S\) are directed trace structures, \(A\) is an alphabet, and \(t \mid A\) denotes the trace \(t\) projected on \(A\), i.e. the trace obtained from \(t\) by deleting all symbols not in \(A\). Concatenation of traces is denoted by juxtaposition, and \((tR)^*\) denotes the set of all finite-length concatenations.
of traces in $tR$. Similarly, $(R)^n$ is defined by: $(R)^1 = R$, and $(R)^{n+1} = R;(R)^n$.

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

Let $U$ be a sufficiently large set of symbols. The characters $e$, $\emptyset$, $b?$, $b!$ and $!b?$ with $b \in U$, are called atomic directed commands. They represent the atomic directed trace structures $(0,0,\{e\})$, $(0,0,\emptyset)$, $(\{b\},\emptyset,\{b\})$, $(0,\{b\},\{b\})$, and $(\{b\},\{b\},\{b\})$ respectively. A directed command is an atomic directed command or an expression for a directed trace structure constructed from atomic directed commands and a finite number of applications of the operations above. In such an expression parentheses are allowed.

In the following, directed commands are denoted by capital E's. The input alphabet, output alphabet, and trace set of the directed trace structure represented by command $E$ are denoted by $iE$, $oE$, and $tE$ respectively. In order to reduce the number of parentheses, we stipulate the following priority rules for the operations just defined. Unary operators (pref, $[,]$, $\uparrow A$) have highest priority. Of binary operators, weaving $(\downarrow)$ has highest priority, then concatenation $(;)$, and finally union $(\cup)$.

Specifications and States

The communication behaviour between a component and its environment can be specified by a prefix-closed, non-empty, regular directed trace structure $R$ with $iR \cap oR = \emptyset$, or, similarly, by a directed command representing such a trace structure.

The valuation of the symbols of a directed command $E$ is a function from $aE$ to $\{0,1\}$. In the range, '0' corresponds to 'low', or 'false', and '1' corresponds to 'high', or 'true'.

The functional operation of a component can be specified by its communication behaviour accompanied by an initial valuation of its symbols. Usually this initial valuation is 'all symbols have value 0', i.e. 'all wires are low', in which case we often omit the initial valuation, and call a directed command (that represents a prefix closed, non empty, directed regular trace structure $R$ with $iR \cap oR = \emptyset$) the specification of (the functional operation of) the component.

Let a component be specified by directed command $E$ and initial valuation $v$. The state of a symbol can be either '0' or '1'. Initially the state of a symbol equals the initial valuation of that symbol. Each occurrence of a symbol in the current trace changes the state of that symbol. Hence, the state of symbol $b$, with $t$ the current trace, equals $v(b)$ iff the length of $t \uparrow \{b\}$ is even. The state of a symbol is denoted by the symbol itself, and instead of $a=0$ and $a=1$ we also write $\neg a$ and $a$ respectively.

The global state of all symbols of a component is given by a valuation.
The directed commands we consider satisfy the following three conditions:
A1 they represent a prefix-closed, non-empty, regular directed trace structure \( R \) with \( \text{iR} \cap \text{oR} = \emptyset \),
A2 they are accompanied by an initial valuation,
A3 a valuation that can occur during operation of the component corresponds to exactly 1 state of the represented trace structure (cf. 'state-graph' in [JE1, sect. 1.1]).

Conditions A1 and A2 are explained above, condition A3 is explained below.

Remark on A3
Condition A3 is not a severe restriction, since each directed command satisfying A1 and A2 can, by introduction of internal variables, be rewritten in an equivalent directed command that satisfies A3. For instance, command \( \text{pref} [a?;b!;a?;b!;a?;c!;a?;c!] \) can, by introduction of internal variable \( x \), be rewritten into the command:
\[
\text{pref} [a?;b!;a?;\bar{x}?;b!;a?;c!;a?;\bar{x}?;c!].
\]

For a component specified by a directed command satisfying A1-3 we can now define the state of a component as the global state of its symbols. Notice that, as a result of A3, the notion 'state' defined here is a proper extension of the notion 'state' of the represented trace structure.

Example
Let a component be specified by:
\[
E = \text{pref} [a?;b!]
\]
initially all symbols have value 0

With \( aba \) as the current trace, the state of the component is characterized by \( \neg a \land b \).

Conditional Transitions
A transition on symbol \( b \) in a directed command is denoted by \( b? \), \( b! \) or \( !b? \) if \( b \) is an input symbol, output symbol, or internal symbol respectively. Such a transition can either be an upgoing transition, if the state immediately preceding the transition satisfies \( b=0 \), or a downgoing transition, if the state immediately preceding the transition satisfies \( b=1 \).

Upgoing transitions on \( b \) can be denoted by \( b\uparrow ? \), \( b\uparrow ! \) or \( !b\uparrow ? \), and downgoing transitions on \( b \) can be denoted by \( b\downarrow ? \), \( b\downarrow ! \) or \( !b\downarrow ? \).

Conditional upgoing transitions on \( b \), denoted by \( \bar{b}\uparrow ? \), \( \bar{b}\uparrow ! \) or \( !\bar{b}\uparrow ? \), and conditional downgoing transitions on \( b \), denoted by \( \bar{b}\downarrow ? \), \( \bar{b}\downarrow ! \) or \( !\bar{b}\downarrow ? \), are defined by:
\[
\bar{b}\uparrow ? = \text{if } b=0 \longrightarrow b? || b=1 \longrightarrow \varepsilon \text{ fi}
\]
\[
\bar{b}\downarrow ? = \text{if } b=0 \longrightarrow \varepsilon || b=1 \longrightarrow b? \text{ fi}
\]

Similar for \( \bar{b}\uparrow ! \), \( \bar{b}\downarrow ! \), \( !\bar{b}\uparrow ? \), and \( !\bar{b}\downarrow ? \).

Notice that (conditional) transitions are defined only in directed commands that specify components, since the notion 'state' is important for their definition.
Example

The specification \( \text{pref } [ a?; b!| x!; | c?; d!| x! ] \), with initially all symbols 0, equals, without using conditional transitions: \( \text{pref } [(c?; d!); a?; b!| x!; [a?; b!]; c?; d!| x!] \).

At a lower level of the derivation (a CMOS implementation of a component instead of a decomposition into (smaller) components), it seems more convenient to reason in terms of states than to reason in terms of transitions (cf. the form of p.r. sets: section 1.3). We feel that the conditional transition is, therefore, a suitable notion.

Compared to the upgoing transition \( b \uparrow \) the conditional upgoing transition \( b' \uparrow \) has the advantage that the state immediately preceding the transition need not, in case of \( b' \uparrow \), satisfy \(-b\). For both \( b \uparrow \) and \( b' \uparrow \) the state immediately following the transition satisfies \( b \). Similar for other conditional transitions.

The usage of conditional transitions is illustrated in section 2.2.3.

A Heuristic for finding Decompositions

The heuristic for finding decompositions given in [JE3] can be summarized as follows (cf. [JE3,p10]). Let the component to be decomposed be specified by \( E.0 \).

1. Partition the outputs of \( E.0 \) into the sets \( B.i \), for \( 1 \leq i < n \).
2. Let \( A.i \) be the set of symbols that immediately precede symbols from \( B.i \) in \( E.0 \).
3. Let \( E.i = (E.0) \cap (A.i \cup B.i) \), where the elements of \( A.i \) are the inputs in \( E.i \) and the elements of \( B.i \) are the outputs in \( E.i \).
4. Verify whether \( E.0 \longrightarrow (i : 1 \leq i < n : E.i) \), where \( \longrightarrow \) means 'can be decomposed into'.
5. If step 4 does not succeed or one of the components \( E.i \), \( 0 \leq i < n \), is not a basic component, try a different partitioning of the outputs of \( E.0 \), or use instead of \( E.0 \) an expansion of \( E.0 \) and apply the above steps again.

Remarks on the heuristic

- In general, there are several ways to solve the problem described in step 5:
  (1) Choose another partitioning of output symbols
  (2) Extend the set of symbols corresponding to output set \( B.i \), for \( 1 \leq i < n \).
      For instance, let \( A.i \) be the set of symbols that the symbols of \( B.i \) depend on.
  (3) Expand \( E.0 \), i.e. introduce one (or more) internal symbols.
- As an aid for step 5 one can investigate the instances of symbols that enable elements of \( B.i \) and those that disable elements of \( B.i \). This may give a lead on how to choose different sets \( A.i \) (see (2)), or how to expand \( E.0 \) (see (3)). This will be elucidated in section 2.2.
1.3 Production Rule Sets to derive (C) MOS Implementations

The production rules used in [AM1,2] specify basic components. They can be derived from expressions similar to directed commands. Martin uses production rules to derive implementations in terms of basic components for circuits specified by such expressions.

In [CvB1] these production rules are used to derive CMOS implementations, where CMOS circuits are viewed as networks of transistors and sources.

This section introduces production rules to derive (speed independent) CMOS implementations for components specified by directed commands satisfying A1-3 (section 1.2, spec. and states).

The notion 'production rule' is extended in such a way that -a- the freedom of implementation given by the specification can be used, -b- calculation with production rules can be done in a convenient way, and -c- a translation of production rules into MOS circuits can be formalized.

By giving a strict translation of production rules into MOS circuits we hope to be able to formalize calculation rules for p.r. sets and prove correctness of the translation (w.r.t. the operational interpretation of p.r. sets) and of the calculation rules (possibly with the help of a suitable switch-level circuit model, as, e.g., given in [Bry]).

This section does not give a complete theory; it gives the basic ideas, analyses what is necessary to complete such a theory, and points out some of the difficulties.

Boolean Terms and Literal Terms

Let \( V \) be a set of variables, with typical elements \( a, b, c, x, y, z \). Let \( X \) be defined as \( X = V \cup \{0,1\} \). Boolean terms are formed from elements of \( X \), the binary infix operators \( \land \) and \( \lor \), and the unary operator \( \neg \) in the usual way. Boolean terms where the negation is placed only at the elements of \( X \) are called literal terms. Literal terms where all occurring elements of \( X \) are unnegated are called positive literal terms; literal terms where all occurring elements of \( X \) are negated are called negative literal terms.

The sets of Boolean terms, literal terms, positive literal terms, and negative literal terms are called \( BT \), \( LT \), \( PLT \), and \( NLT \) respectively.

Theorem 1

With the transformation rules\(^2\): \( \neg \neg f = f \), \( \neg (f \lor g) = \neg f \land \neg g \), and \( \neg (f \land g) = \neg f \lor \neg g \), each boolean term can be transformed into an equivalent literal term. Although more transformation paths are possible, all transformation paths of a specific boolean term \( f \) lead to the same literal term.

The proof of theorem 1 is given in Appendix A.

\(^2\) A transformation rule has the form \( D = E \) where \( D \) and \( E \) are general forms of terms. Such a transformation rule means that wherever a term of form \( D \) appears (even as a subterm), it may be replaced by the corresponding form \( E \).
The literal term resulting from transformations of boolean term \( f \) with the rules above is called \( f \)'s corresponding literal term, and is denoted by \( \text{lt}(f) \).

Syntactical equality of Boolean terms is denoted by \( = \). Semantical equivalence is denoted by \( '=\)'. The usual semantics for terms is used, where \( \wedge, \vee, \neg, '0', \) and \( '1' \) stand for 'AND', 'OR', 'NOT', 'false', and 'true' respectively.

Properties (see Appendix A)

1. \( LT \subseteq BT \), \( PLT \subseteq LT \), and \( NLT \subseteq LT \)
2. \( f \in BT \Rightarrow f = \text{lt}(f) \)
3. \( f \in LT \Rightarrow f \equiv \text{lt}(f) \)
4. \( \text{lt}(f) \in PLT \Leftrightarrow \text{lt}(\neg f) \in NLT \)

Production Rules

CMOS circuits can be viewed as networks of transistors and sources. To derive CMOS implementations of components we use production rules. Production rule sets specify the transitions on the output symbols of a component, and can easily be inferred from the component's specification as a directed command satisfying A1-3.

In this subsection we define production rule sets (which are extensions of the production rule sets defined in [AM1,2]), and we show how to infer production rule sets from a directed command satisfying A1-3.

A production rule for a symbol \( b \) has the form \( G \rightarrow P \) where the guard \( G \) is a boolean term, and the production \( P \) either equals 'dc' (which stands for "don't care"), or is an assignment of the form \( b:=E \), where \( E \) is a boolean term.

A production rule set (p.r. set for short) for symbol \( b \) is a non-empty set \( \{ i : 1 \leq i \leq n : G_i \rightarrow P_i \} \) of production rules for symbol \( b \) satisfying:

1. \( (G_1 \lor G_2 \lor ... \lor G_n) = 1 \).
2. If \( (G_i \land G_j) \neq 0 \), for some \( i,j : 1 \leq i < j \leq n \),
   then either \( P_i \equiv dc \) and \( P_j \equiv dc \),
   or \( P_i \equiv (b:=E_i) \) and \( P_j \equiv (b:=E_j) \) where \( (G_i \land G_j) \equiv (E_i=E_j) \) is invariantly true.
3. a) If a production rule has the form \( f \rightarrow b:=e \) with \( e \in \{0,1\} \) and \( f \) holds in a computation, then \( f \) must be stable in this computation until \( b = e \).
   b) If a production rule has the form \( f \rightarrow b:=e \) with \( e \in \{0,1\} \), then the production rules \( f \land e \rightarrow b:=1 \) and \( f \land \neg e \rightarrow b:=0 \) must satisfy -3-a.

The notion 'invariantly' is used w.r.t. the component's communication behaviour, i.e. 'P is invariantly true' means: 'all reachable states of the component satisfy P'. Notice that -2- and -3- correspond to Martin's Non-Interference and Stability requirement ([AM2]) respectively.
The operational interpretation of p.r. sets is the following: If the state of a component satisfies $G_i$ (for some $i$), and the corresponding $P_i$ is an assignment, then $P_i$ must be executed (possibly with some delay). If $P_i$ equals $dc$, we have the freedom to choose (implement) any assignment. It will often be the case, if $P_i$ equals $dc$, that the component cannot reach a state satisfying $G_i$.

To specify a component by means of p.r. sets we need to give a p.r. set for each output symbol. A p.r. set for an output symbol $b$ can be found by characterizing the states of the component that enable a transition on $b$, characterizing the states in which $b$ must remain its value (production $b := b$), and those that do not occur during operation of the component (production $dc$). For components specified by a directed command satisfying $A1$-$3$, states are defined in the previous section. Notice that characterizing these states by means of a predicate on the component's symbols is possible as a result of $A3$.

Notice also that a p.r. set only specifies the transitions on output symbols; it gives no information about transitions on input symbols (a concern of the component's environment), like a directed command does.

Remarks on the definition of p.r. sets

R1 In order to specify a component the production rules of a p.r. set may not be self-invalidating ([AM2]). A production rule is called self-invalidating if it has the form $G \leftarrow b := E$ where $(b = E) \neg \Rightarrow G$.

R2 Conditions -2-, -3-, and absence of self-invalidating production rules imply that a p.r. set defines a set of stable states for the components, viz. those where $G \Rightarrow (b = E)$ for all production rules of the form $G \leftarrow b := E$ (see also remark R4). A change on inputs may cause instability, but on account of -3- the next input change causing instability arrives after stability is reached. Note that the 'stability requirement' therefore resembles the well-known 'fundamental mode assumption' (defined in, e.g. [JE1,2]).

Remarks on the notation

- A p.r. set $\{i : 1 \leq i \leq n : G_i \leftarrow P_i\}$ is denoted by:
  $$\begin{align*}
  G_1 & \leftarrow P_1 \\
  G_2 & \leftarrow P_2 \\
  \vdots
  \end{align*}$$

- A p.r. set of the form $\{\text{true} \leftarrow P\}$ is shortened as $P$.

- If a component is specified by more than one p.r. set, the p.r. sets are connected by ampersands ('&'). Since the order is irrelevant, we call p.r. sets connected by ampersands a bag of p.r. sets (non-empty by definition).
Example
For the component specified by:
\[ E = \text{pref} \{ a?;b! \} \]
initially all symbols have value 0
we have:
\[
\begin{align*}
-a \land -b & \rightarrow b := 0 \\
a \land -b & \rightarrow b := 1 \\
a \land b & \rightarrow b := 1 \\
-a \land b & \rightarrow b := 0 \\
\end{align*}
\]
\[
\begin{align*}
&\{ a \rightarrow b := 1 \\
&-a \rightarrow b := 0 \\
&\text{true} \rightarrow b := a \\
\end{align*}
\]

Translation of P.R. Sets into (C) MOS Circuits
In order to satisfy the operational interpretation given before, the translation of a production rule of the form 'G \rightarrow x := y', with \( x \in V \) and \( y \in X \), must be a transistor network connecting \( x \) and \( y \) if and only if \( G \) holds. To be able to give the translation of p.r. sets into MOS circuits, we start by giving the translation of boolean terms into transistor networks. This translation is basically the same as the translation from boolean terms into relay contact networks given in [Mil, sect. 2.5].

Translation of Boolean Terms into Transistor Networks
A boolean term \( f \) corresponds to a transistor network called \( TN(f) \). \( TN(f) \) has two external nodes, that are connected if and only if \( f \) holds.
A boolean term \( f \) corresponds to the same network as \( \lt(f) \). For literal terms we define the corresponding transistor networks below, where \( v \in V, x \in X, \{f,g\} \subseteq LT \):
\[
\begin{align*}
0 & \quad \text{no connection between the external nodes (fig. 1.1a)} \\
1 & \quad \text{permanent connection between the external nodes (fig. 1.1b)} \\
v & \quad \text{one n-transistor with gate} \, \, v \, \, \text{and as pass nodes the external nodes (fig. 1.1c)} \\
-x & \quad \text{one p-transistor with gate} \, \, x \, \, \text{and as pass nodes the external nodes (fig. 1.1d)} \\
f \lor g & \quad \text{parallel connection of} \, \, TN(f) \, \, \text{and} \, \, TN(g) \, \, \text{(fig. 1.1e)} \\
f \land g & \quad \text{serial connection of} \, \, TN(f) \, \, \text{and} \, \, TN(g) \, \, \text{(see fig. 1.1f)}
\end{align*}
\]

\[ a : TN(0) \quad b : TN(1) \quad c : TN(v) \quad d : TN(-x) \quad e : TN(f \lor g) \quad f : TN(f \land g) \]
figure 1.1 : Transistor networks corresponding to literal terms.
Notice that the transistor networks corresponding to positive (negative) literal terms consist of n-transistors (p-transistors) only.

Translation of P.R. Sets into MOS Circuits
Production rule sets correspond to MOS circuits. For \( f \in BT \) the translation of a production rule for symbol \( x, x \in V \), into a circuit part is defined as follows:

- Production rule \( f \rightarrow dc \) corresponds to an empty circuit part.
- Production rule \( f \rightarrow x:=e' \), with \( e \in X \), corresponds to a circuit part consisting of \( TN(f) \), where one of the external nodes is connected to \( e \) and the other is connected to \( x \).
- Production rule \( f \rightarrow x:=e' \), with \( e \notin X \), corresponds to the circuit part formed by the two circuit parts corresponding to the rules: \( f\&e \rightarrow x:=1' \) and \( f\&e \rightarrow x:=0' \).

A (bag of) p.r. set(s) is translated into the circuit consisting of the circuit parts translated from its production rules, where all nodes labelled with the same element of \( X \) are connected.

Remark
R3 In order to represent a Complementary MOS circuit a p.r. set must have the form:

\[
\begin{align*}
\triangleright & \quad \begin{cases} f \rightarrow x := 1 \\ g \rightarrow x := 0 \end{cases} \quad \text{with } \text{lt}(f) \in NLT \text{ and } \text{lt}(g) \in PLT, \text{ or} \\
\triangleright & \quad \begin{cases} f \rightarrow dc \\ g \rightarrow x := 1 \end{cases} \quad \text{with } \text{lt}(f) \in NLT \text{ and } \text{lt}(g) \in PLT, \text{ or} \\
\triangleright & \quad x := f \quad \text{with } \text{lt}(f) \in NLT \text{ or } f \in X \quad \text{(cf. property (4) and the translation of p.r. sets).}
\end{align*}
\]

Calculating with P.R. Sets
The p.r. sets derived from a directed command satisfying A1-3 need not necessarily be translated (with the translation above) into a correctly functioning speed independent (Complementary) MOS implementation of the component. Therefore, calculation (or transformation) rules are needed to find p.r. sets representing such an implementation, preferably one that is as cheap as possible (for costs measures: see sect. 1.4).

This subsection gives some of the requirements such calculation rules must satisfy, and analyses the possibilities and difficulties of calculation rules. Finally, some calculation rules are given and some examples to illustrate them.

A calculation (transformation) rule for a (bag of) p.r. set(s) must be such that:

a each element of the resulting bag is a p.r. set, i.e. it satisfies -1-, -2-, and -3-.
b the result specifies or implements the same component (see R4 below).
c the transformation does not introduce self-invalidating production rules.
The resulting p.r. sets of a calculation must represent a correctly functioning speed independent (Complementary) MOS implementation of the component. How to determine whether a p.r. set represents such an implementation still is a subject for further research. 'Correctly functioning' means, except from -1-, -2-, -3-, and the absence of self-invalidating production rules, that feedback loops are implemented correctly. A minimum, but insufficient, requirement on p.r. sets in order to have correct feedback-loops, is that \( b \) does not occur in \( G \) and \( E \) of a production rule of the form \( G \rightarrow b := E \). Notice that, as a result of this, self-invalidating production rules do not occur. For 'Complementary'; see remark R3. Speed independence is implied by -2-, -3-, and the absence of self-invalidating production rules only if during calculation no internal variables are introduced. Checking speed independence is difficult if internal variables are introduced during calculation. There are two ways to attack this problem: (A) prove the speed independence of the resulting p.r. set, or (B) find calculation rules introducing internal variables that preserve speed independence. Both (A) and (B) might be possible with the help of a suitable (switch level) circuit model (e.g. finding general requirements for the resulting p.r. set), or with the help of decompositions.

R4 During calculation with a p.r. set the stable states defined by this p.r. set (cf. R2) are allowed to change only by filling in the \( dc \). Recall that a stable state satisfies \( G \Rightarrow (b = E) \), for each production rule \( G \rightarrow b := E \). The freedom of implementation is placed in the production rules of the form \( H_i \rightarrow dc \). Of course, this freedom is also available in cases where \( G_i \wedge G_j \wedge \neg (E_i = E_j) \) for production rules \( G_i \rightarrow b := E_i \) and \( G_j \rightarrow b := E_j \), since, on account of -2-, no reachable states satisfy this predicate. Therefore, two predicates on p.r. sets are defined as follows:

\[
\text{SPI}(prs) = (Ai : 1 \leq i \leq n : G_i \Rightarrow (b = E_i)) \lor H
\]

\[
\text{SP2}(prs) = (Ai : 1 \leq i \leq n : G_i \Rightarrow (b = E_i)) \wedge \neg H
\]

where \( prs = \{i : 1 \leq i \leq n : G_i \rightarrow b := E_i\} \cup \{j : 1 \leq j \leq m : H_j \rightarrow dc\} \).

For bags of p.r. sets, the predicates are defined as the conjunction of the predicates applied on the elements:

\[
\text{SPi}(prs_1 \& prs_2) = \text{SPI}(prs_1) \land \text{SPI}(prs_2)
\]

where \( prs_1 \) is a p.r. set, \( prs_2 \) is a bag of p.r. sets, and \( i \in \{1,2\} \)

For bags of p.r. sets equality and implication are defined as follows:

\[
(prs_1 = prs_2) = (\text{SPI}(prs_1) = \text{SPI}(prs_2)) \land (\text{SP2}(prs_1) = \text{SP2}(prs_2))
\]

\[
(prs_1 \Rightarrow prs_2) = (\text{SPI}(prs_1) \Rightarrow \text{SPI}(prs_2)) \land (\text{SP2}(prs_1) \Rightarrow \text{SP2}(prs_2))
\]

where \( prs_1 \) and \( prs_2 \) are bags of p.r. sets.
P.r. sets \( prsl \) are said to be implemented by p.r. sets \( prs2 \) if \( prsl \subseteq prs2 \). Condition b means that \( prsl \) may be transformed into \( prs2 \) only if \( prsl \subseteq prs2 \).
\( \square \) (end remark R4)

**Calculation rules**
The following calculation rules do not form a complete set. For practical purposes more calculation rules will (probably) be needed. As explained above, finding calculation rules introducing internal variables is difficult. Conditions a, b (with R4!) and c, however, form a good guide for finding calculation rules.

\[ \text{CR1a} \quad \text{production rule} \quad f_0 \rightarrow x := e_0 \quad \text{may be replaced by} \quad f_1 \rightarrow x := e_1 \quad \text{if} \quad f_0 = f_1 \quad \text{and} \quad e_0 = e_1. \]

\[ \text{CR1b} \quad \text{production rules} \quad f_0 \rightarrow R \quad \text{and} \quad f_1 \rightarrow R \quad \text{may be replaced by} \quad f_0 \lor f_1 \rightarrow R. \]

\[ \text{CR1c} \quad \text{production rule} \quad f \rightarrow x := e \quad \text{may be replaced by} \quad f \land e \rightarrow x := 1. \]

Calculation rules \( \text{CR1a,b,c} \) may also be used in the opposite direction.

\[ \text{CR2a} \quad \text{production rule} \quad h \rightarrow dc \quad \text{may be replaced by} \quad h \rightarrow x := e. \]

\[ \text{CR2b} \quad \text{p.r. set} \quad \begin{cases} f_0 \rightarrow x := 1 \\ f_1 \rightarrow x := 0 \end{cases} \quad \text{may be replaced by} \quad \begin{cases} f_0 \land \neg f_1 \rightarrow x := 1 \\ f_0 \land f_1 \rightarrow x := 0 \\ f_0 \lor f_1 \rightarrow dc \end{cases}. \]

For these calculation rules conditions a, b, and c are easily verified. Application of rules \( \text{CR1a,b,c} \) and \( \text{CR2b} \) lead to equality of the p.r. sets involved (see R4), which explains the bidirectionality of these rules. Rule \( \text{CR2a} \) uses the freedom given by the specification.

\[ \text{CR3} \quad \begin{cases} i : 1 \leq i \leq n : G_i \rightarrow x := E_i \cup \{ j : 1 \leq j \leq m : H_j \rightarrow dc \} \\ \neg \{ i : 1 \leq i \leq n : G_i \rightarrow z := \neg E_i \cup \{ j : 1 \leq j \leq m : H_j \rightarrow dc \} \\ \& \{ z \rightarrow x := 0, \neg z \rightarrow x := 1 \} \end{cases} \]

For \( \text{CR3} \) conditions a, b, and c also are easily verified.

The proof that \( \text{CR3} \) preserves speed independence can be given with -abstract- decomposition: Let the starting p.r. set of \( \text{CR3} \) correspond to directed command \( E \). Replacing all occurrences of \( x! \) in \( E \) by \((!z;x!)/(!z;x!) \) leads to an expansion \( EE \) of \( E \). Consequently \( E \rightarrow EE \). Projecting \( EE \) on \((aE \cup \{z!/\})\{x!/\} \) gives a directed command, say \( E' \), that is similarly formed as \( E \) (only \( x! \) replaced by \( z! \)). Projecting \( EE \) on \((z!, x!)\) gives \( \text{pref}[z!?;x!] \). It is now easily verified that \( EE \rightarrow E', \text{pref}[z!?;x!] \), and consequently \( E \rightarrow E', \text{pref}[z!?;x!] \). Choosing the initial valuation of \( z \) as \( z \neq x \) completes the proof.

Calculation rules introducing internal variables, like \( \text{CR3} \), can be used for two purposes: to correct a feedback of the specified output; or to regain a complementary implementation. For the latter purpose we use the information given in remark R3.
R5 In case \( \neg(F \land G) \) the following useful calculation rule can be derived with the calculation rules above:

\[
\begin{align*}
F & \leftarrow x := 1 \\
G & \leftarrow x := 0 \\
\neg F \land \neg G & \leftarrow x := x
\end{align*}
\]

\( = \) \{using CR1b,c\}

\[
\begin{align*}
F \lor (\neg F \land \neg G \land x) & \leftarrow x := 1 \\
G \lor (\neg F \land \neg G \land \neg x) & \leftarrow x := 0
\end{align*}
\]

\( = \) \{using CR1a, predicate calculus, and \( \neg(F \land G) \)\}

\[
\begin{align*}
F \lor (\neg G \land x) & \leftarrow x := 1 \\
\neg F \land (G \land \neg x) & \leftarrow x := 0
\end{align*}
\]

\( = \) \{using CR1a,c\}

\[
\begin{align*}
x & := F \lor (\neg G \land x)
\end{align*}
\]

\( = \) \{using CR3 \}

\[
\begin{align*}
z & := \neg F \land (G \land \neg x) \land x := \neg z
\end{align*}
\]

Example 1.1

Let a component be specified by:

\[
\text{pref } [(y_0?;x^t)^2 \mid (y_1?;x^t)^2], \text{ with initially all variables } 0
\]

The first p.r. set in the following calculation can be inferred directly from this specification. The last p.r. set of the calculation is translated into the usual CMOS OR-gate implementation depicted in figure 1.2a. The calculation rules used are CR2a, CR1a,b and CR3 respectively. Notice that CR3 is used to derive a complementary implementation (cf. also R3).

\[
\begin{align*}
\neg y_0 \land \neg y_1 & \leftarrow x := 0 \\
\neg y_0 \land y_1 & \leftarrow x := 1 \\
y_0 \land \neg y_1 & \leftarrow x := 1 \\
y_0 \land y_1 & \leftarrow \text{dc}
\end{align*}
\]

\( \models \)

\[
\begin{align*}
\neg y_0 \land \neg y_1 & \leftarrow x := 0 \\
\neg y_0 \land y_1 & \leftarrow x := 1 \\
y_0 \land \neg y_1 & \leftarrow x := 1 \\
y_0 \land y_1 & \leftarrow x := 1
\end{align*}
\]

\( = \) \{using CR1a,b\}

\[
\begin{align*}
\neg y_0 \land \neg y_1 & \leftarrow x := 0 \\
y_0 \lor y_1 & \leftarrow x := 1
\end{align*}
\]

\( = \) \{using CR3 \}

\[
\begin{align*}
\neg y_0 \land \neg y_1 & \leftarrow z := 1 \\
y_0 \lor y_1 & \leftarrow z := 0 \land \neg z \leftarrow x := 1 \\
z & \leftarrow x := 0
\end{align*}
\]

More examples of the use of production rule sets and calculation rules are given in chapter 2. From example 1.1 and the examples in chapter 2 one might get the impression that calculation rules are superfluous since calculation is rather simple. Therefore the following example is given where a calculation leads to an implementation that is not speed independent.
Example 1.2

Consider the component specified by \texttt{E3.0} from section 2.2.2. The specification is:

\[ \text{pref } [ (y_0?; a?; a_0)^2 | (y_0?)^2 | (a?)^2 ] \text{, initially all variables are } 0. \]

The first p.r. set in the calculation below implements \texttt{E3.0} (cf. sect. 2.2.2). The calculation leads to the implementation depicted in figure 1.2b. This implementation is not speed independent: consider trace \( y_0 \ y_0 \ a \) and suppose \( z \) has changed to 1 after \( y_0 \) is received, but, due to delay in the OR-gate, \( z \) has not yet changed back to 0 when \( a \) is received. It is now possible that \( a_0 \) changes (unintendedly) to 1.

Notice that only the step introducing \( z \) is incorrect. In section 2.2.2 a correct implementation of \texttt{E3.0} is derived.

\[
\begin{align*}
-a & \longrightarrow a_0 := 0 \\
-y_0 \land a & \longrightarrow a_0 := 1 \\
-y_0 \land a & \longrightarrow a_0 := a_0 \\
\implies a_0 & := (y_0 \land a) \lor (-y_0 \land a \land a_0) \\
\implies a_0 & := a \land (y_0 \lor a_0) \\
\implies z & := (y_0 \lor a_0) \land a_0 := (a \land z)
\end{align*}
\]

\textbf{Concluding Remarks on p.r. sets}

The direct translation has the disadvantage that some (tricky) implementations, e.g. the one in example 1.6 (sect. 1.4), cannot be represented by p.r. sets. By giving a direct and simple translation, however, we hope to simplify the correctness proofs of the translation and of the calculation rules.

In the given examples, working with p.r. sets seems to be very convenient. An additional advantage of using p.r. sets for deriving implementations is that we might profit from work done on boolean term optimization (e.g. in [Br1,2]).

See also chapter 3: Concluding remarks.
1.4 Cost Measures for (C) MOS Implementations

In order to compare different implementations of a circuit (or a component), cost measures are required. The most important cost measures for circuit implementations are measures for size and delay. Since the descriptions of the circuit implementations we give are relatively abstract (connections of transistors instead of, e.g., layout), the cost measures we give are abstract too. For our purpose, i.e., comparing different implementations, abstract measures suffice.

In this section we give size and delay measures for implementations of cells (for 'cell': see section 1.1). The contributions of connection wires between cells to size and delay are not taken into account. For the size measure this is not a severe disadvantage, since the contribution of wiring to size forms a fixed percentage of the total circuit size, and can therefore be neglected when the measures are used for comparing only. For an abstract description as we use, the additional delays caused by connection wires are impossible to estimate. We therefore accept a less accurate delay measure that ignores them.

Size Measure

Our size measure is related to the gate matrix layout style ([W&E], sect.7.3). Therefore we first give a short – and abstract – introduction to this layout style.

Gate matrix layout

When using gate matrix layout to implement cells, the cell area is rectangular. At the top of the cell area a (horizontal) row connected to the high voltage source (Power, or 1) is placed, and at the bottom of the cell area a row connected to the low voltage source (Ground, or 0) is placed. The upper half of the cell area is used to implement p-transistors, the lower half is used to implement n-transistors. For each gate node and each external (input/output) node one, or more (see remark 1 below), (vertical) columns are placed. Transistors are implemented on these columns; of course in the correct half of the cell area, and on the columns corresponding to their gate nodes. Horizontal rows are used to connect the pass nodes of a transistor to the corresponding columns or to other pass nodes. To illustrate this we give a simple example.

Example 1.3a

The implementation of the CMOS Inverter is depicted in figure 1.3a. In figure 1.3b its gate matrix layout is (schematically) depicted.
The size of the cell area is roughly determined by (A) the number of columns corresponding to
gate nodes (the width), and (B) the maximal number of transistors placed on such a column
(the height).

There are two reasons for not counting the columns that do not correspond to gate nodes:
firstly, they do not contain transistors and therefore contribute relatively little to the width of the
cell area, and secondly, we also use the size measures to compare sizes of components that form
parts of the cell, and when composing two components the output of one often is a gate node of
the other and hence, the "output column" (which mostly is the only column that does not
 correspond to gate nodes) is already counted as a "gate node column".

As a third — and least important — part of the size measure we use the number of transistors,
since this gives an impression of the amount of compression of the cell area that is possible in
the actual layout.

As size measure we now have \((w,h,t)\), where \(w\), the width of the cell area, corresponds to (A)
above; \(h\), the height of the cell area, corresponds to (B) above; and \(t\) is the number of
transistors. Notice that a size measure \((w,h,t)\) satisfies: \(t \leq wh\).

If each gate node corresponds to exactly one column, \(w\) equals the number of gate nodes in the
cell, and \(h\) equals the maximal number of gates that are controlled by one gate node.

An implementation with size measure \((w_0,h_0,t_0)\) is smaller than one with size measure
\((w_1,h_1,t_1)\) if the size measures satisfy:
\[
((w_0,h_0) < (w_1,h_1)) \lor ((w_0,h_0) = (w_1,h_1) \land t_0 < t_1),
\]
where \((w_0,h_0) \leq (w_1,h_1) \iff w_0 \leq w_1 \land h_0 \leq h_1\).

Remarks

1. It is allowed for a gate node to correspond to more than one column. Of course, these
columns must, in such case, be connected, which can be realized by a horizontal row. This
may be profitable if, for instance, one of the columns is twice as high (has two times the
number of transistors) as the other columns. Using two columns instead of this one
halves the height of the cell area, whereas the width is extended only with one. Notice
that, as a result hereof, it is often possible to give several sizes for one cell area, from
which we may choose the most convenient one (e.g. for comparing it is convenient to
choose sizes with equal heights).

2. Composing two parts with sizes \((w_0,h_0,t_0)\) and \((w_1,h_1,t_1)\) in one cell leads to a cell area size
\((w,h,t)\) with \((w,h) \leq (w_0+w_1,\max(h_0,h_1))\) and \(t = t_0+t_1\).

To calculate the size of a cell area it is not sufficient to know the sizes of the parts of the
cell. For several reasons the internal structure of these parts is important:
- The cell area has one height: the heights of the parts must be adjusted, which may
lead (see remark 1) to a change of width.
Some nodes are gate nodes in more than one part: it may be possible to use less columns than the total number of corresponding columns in the parts without changing the height.

Example 1.3b

The CMOS implementation of the Inverter (see fig. 1.3a), contains two transistors, and they are controlled by the same node. As a result of this the gate matrix layout of the Inverter (see schematic fig. 1.3b) contains only one "gate node column", and the number of transistors on this column is 2. The size of the Inverter therefore is (1,2,2).

Delay Measure

A fair estimate of the propagation delay can be made by using the RC time constants of the circuit (cf. [Muk], sect. 6.5, and [W&E], sect. 4.3.5). The delay is measured at the gate nodes and output nodes as follows.

- The resistance of a node to a source is defined by regarding the transistors that are passed on the path from the source to the node. Note that we need only take those paths into account that conduct when a transition at the node occurs (i.e. the paths that conduct when the node is being (dis)charged), since only these paths are causing a delay. A single transistor has resistance 1, and the resistance obeys the following—well known—rules:

\[ R = R_1 + R_2 \]

for serial connection of resistances \( R_1 \) and \( R_2 \);

\[ \frac{1}{R} = \frac{1}{R_1} + \frac{1}{R_2} \]

for parallel connection of resistances \( R_1 \) and \( R_2 \).

The resistance of a node is defined as the maximum of the node's resistance to the high voltage source and its resistance to the low voltage source.

- Capacitance of a node is measured by the number of gates it controls. Since the gates usually come in pairs (one of an n-transistor and one of a p-transistor), we define the capacitance of a node as the number of such n/p-pairs the node controls.

For the output node we assume an additional capacitance of 1, which means we assume that the output node controls 1 n/p-pair of another component. Connecting components may, therefore, change the capacitance of one of the output nodes, if that output node does not control exactly 1 n/p-pair of the other component. Wires that connect distinct cells often require a larger capacitance, which then causes inaccuracy of the given delay measure (see above).

- The delay caused by a gate node or an output node is defined as the product of its resistance and its capacitance. For a cascade of elements we add the individual delays, and for elements connected in parallel we take the maximum of the individual delays.
Example 1.3c
For the Inverter (fig. 1.3a) the only node contributing to the delay is the output node, since the only other node is an input node and therefore has no R-value. The output node has resistance 1 to each source, and hence its R-value is 1. The C-value of the output node is 1 because of the additional capacitance for output nodes. As a result of this, the delay estimate of the Inverter is 1.
For a cascade of $k$ Inverters the delay estimate is $k$.

Examples
1.4 The NOR-gate (fig. 1.4a) has two gate (input) nodes, and both of them control two transistors. The number of transistors is 4. Consequently, the size is (2,2,4).
As in example 1.3c, the only node that contributes to the delay estimate is the output node. The R-value of the outputnode is 2, the C-value is 1 (add. cap.). Therefore the delay estimate of the NOR-gate is 2.
1.5 The AND-gate (fig. 1.4b) has three gate nodes: the two input nodes and the internal node that controls the inverter. Each gate node controls two transistors. Its size is (3,2,6).
The delay estimate is 3 (2 (compare with ex. 1.4) + 1 (compare with ex. 1.3c)).
1.6 The implementation of the (Muller) C-element (figure 1.4c: from [CvB1]) contains four gate nodes: $a, b, c,$ and the internal gate node $x$. The maximum number of transistors controlled by one gate node is 4 (by $a$ and $b$). The size of this implementation therefore is (4,4,12).
The nodes that contribute to the delay measure are $x$ and $c$. A transition on $x$ occurs only when $a$ and $b$ are equal and differ from $c$. Therefore, there are (in this implementation) two paths to a source (each with resistance 2) that conduct during (dis-)charging of $x$. As a result the R-value of $x$ is 1. Since both $x$ and $c$ control one n/p-pair within the C-element the C-value of $x$ is 1, and (with the additional capacitance of 1 for output nodes) the C-value of $c$ is 2. It is easily seen that the R-value of $c$ is 1. We can now conclude that the delay estimate of the C-element is 3.

![figure 1.4a: NOR-gate.](image1) ![figure 1.4b: AND-gate.](image2) ![figure 1.4c: C-element.](image3)
2 The Demultiplexer

The demultiplexer is one of the basic components in [B&S]. In this chapter some implementations of the demultiplexer are derived using the techniques described in chapter 1. At the end of the chapter, these implementations are compared using the cost measures from section 1.4.

2.1 Introduction

The demultiplexer is a device that takes care of data transport. It is connected to — two or more — receivers and one sender. The communication between the demultiplexer and its environment is realized by means of the four phase handshaking protocol. The receivers are active, which means that they start up the communication with the demultiplexer. To be precise: a receiver sends a data request after which it will receive the data from the demultiplexer. The sender is passive, which means that the demultiplexer starts up the communication between sender and demultiplexer, also with a data request.

The transport protocol for the demultiplexer is, therefore, as follows: After receiving a data request from a receiver, the demultiplexer sends a data request to the sender. The data, received from the sender, is to be send to the receiver that requested it. Furthermore, as a result of the four phase handshaking protocol, a "return to zero" step is included.

An important restriction — on the receivers — is that only one receiver may be involved in the transport protocol at the same time.

If we abstract from the coding of the data, and represent the data by symbol $d$, the specification of the demultiplexer connected to $M$ receivers, with $2 \leq M$, is:

$$\text{pref} \left[ \{ i : 0 \leq i < M : ( y_i ? ; y_i ! ; d ; d_i )^2 \} \right]$$

initially all symbols have value 0

where $y_i$ is the data request of receiver $i$

$y$ is the data request to the sender

$d$ is the data from the sender, and

$d_i$ is the data to receiver $i$.

Notice that $d$ and $d_i$ are not trace symbols. The exact ordering between them depends on the coding of the data. For instance, if $d?$ stands for $d0?|d1?$ and $d_i?$ stands for $d0i?|d1i?$, then $d?; d_i?$ stands for $(d0?; d0i?)(d1?; d1i?)$. Since the data communication must be delay insensitive, the data must be implemented as a DI-code ([TV]).

So, in general, the demultiplexer is connected to $M$ receivers and the data is implemented as a DI-code. In the next section we consider the demultiplexer with 2 receivers and the data implemented as a double rail code. Double rail coding means using two variables per bit, that
are complementary in case of data transport, and both low otherwise. As a start we consider DMX1: a demultiplexer with two receivers and 1 bit double rail data (see fig. 2.1). DMX1 is specified by \( E_0 \) as follows:

\[
E_0 = \text{pref} \left[ (y_0,y!;a?,a_0!)^2 \mid (y_0,y!;b?,b_0!)^2 \mid (y_1,y!;a!,a_1!)^2 \mid (y_1,y!;b!,b_1!)^2 \right]
\]

In section 2.2 implementations of DMX1 are derived, and the generalizations to N bits double rail coded data are given. In section 2.3 the different solutions found in section 2.2 are compared. The generalization to M receivers and an arbitrary DI code is left to the reader.

In this chapter we do not aim to give formal correctness proofs of the decompositions (see step 4 of the heuristic), but we aim to demonstrate the heuristic for finding decompositions, the use of p.r. sets, and the use of the costs measures given in section 1.4. During decomposition we will bare in mind that the cell structure should be such that generalization is possible in an elegant way (e.g. by adding a number of equally formed cells), and we will regard the design strategy as formulated in section 1.1, viz. global forks must be avoided.

\[
\begin{align*}
\text{sender} & \quad \text{receiver 0} \\
y \quad y_0 & \quad a_0 \\
a \quad a_1 & \quad b_0 \\
b \quad b_1 & \quad y_1
\end{align*}
\]

\[
\text{figure 2.1 : DMX1.}
\]

2.2 Derivations

To derive implementations of DMX1 we start with specification \( E_0 \), and use the heuristic given in section 1.3. Recall:

\[
E_0 = \text{pref} \left[ (y_0,y!;a?,a_0!)^2 \mid (y_0,y!;b?,b_0!)^2 \mid (y_1,y!;a!,a_1!)^2 \mid (y_1,y!;b!,b_1!)^2 \right]
\]

The partitioning of output symbols (step 1 in the heuristic) that we choose is the set of request outputs, \( B_0 = \{y!\} \), and the set of data outputs, \( B_1 = \{a_0!,a_1!,b_0!,b_1!\} \). The sets of immediately preceding symbols (step 2) are \( A_0 = \{y_0,y_1\} \) and \( A_1 = \{a?,b?\} \) respectively.

Applying step 3 we get:

\[
\begin{align*}
E_1 &= E_0 \cap (B_0 \cup A_0) = \text{pref} \left[ (y_0,y!;y!)^2 \mid (y_1,y!;y!)^2 \right] \\
E_2 &= E_0 \cap (B_1 \cup A_1) = \text{pref} \left[ (a?,a_0!)^2 \mid (b?,b_0!)^2 \mid (a?,a_1!)^2 \mid (b?,b_1!)^2 \right]
\end{align*}
\]

Notice that \( E_1,E_2 \) is not a decomposition of \( E_0 \). The reason hereof is that the data outputs, \( a_0!,a_1!,b_0!,b_1! \), do not depend on the data inputs, \( a? \) and \( b? \), only. In fact, every instance of \( a_0! \) is preceded, and therefore enabled, by \( a? \). However, not every instance of \( a? \) precedes, and therefore enables \( a_0! \). It is necessary to distinguish between those instances of \( a? \) that enable \( a_0! \), and those that do not enable \( a_0! \) (cf. sect. 1.2, 'remarks on the heuristic'). Similar for \( a? \) and \( a_1! \), \( b? \) and \( b_0! \), and \( b? \) and \( b_1! \).
Suggestion (1) for step 5 (cf. sect. 1.2, 'remarks on the heuristic') tries to avoid this problem, which is not possible in our example.Suggestions (2) and (3) try in different ways to make the necessary distinction. (2) is applied in solutions 1 and 2 (sect. 2.2.1 and 2.2.2), and (3) is applied in solution 3 (sect. 2.2.3).

2.2.1 Solution 1

Derivation
As before, we choose partitioning \( B_0, B_1 \). The set corresponding to \( B_0 \) is \( A_0 \). Since it is necessary to distinguish the instances of \( a^?, b^? \) following \( y_0^! \) and those following \( y_1^! \), we choose, instead of \( A_1 \), the set of symbols the symbols of \( B_1 \) depend on, which is \( A_2 = \{ y_0^?, y_1^?, a^?, b^? \} \). Instead of \( E_2 \) we now get:

\[
E_3 = E_0 \setminus (B_1 \cup A_2)
\]

Notice that \( E_1, E_3 \) is a decomposition of \( E_0 \). Since the inputs in \( E_3 \) are ordered, \( E_3 \) is not a DI-component. The ordering of inputs however is an obligation of the environment, i.e., \( E_0 \).

We therefore omit the ordering of inputs and obtain, as a candidate for decomposition of \( E_3 \):

\[
E_4 = \text{pref} \left[ (y_0^?; a^?; a_0^!)^2 \mid (y_0^?; b^?; b_0^!)^2 \mid (y_1^?; a^?; a_1^!)^2 \mid (y_1^?; b^?; b_1^!)^2 \right]
\]

Notice that \( E_4 \) is indeed a decomposition of \( E_3 \). Notice also that \( E_1 \) and \( E_4 \) are DI-components. As a consequence, \( E_0 \) can be DI-decomposed into \( E_1, E_4 \). In example 1.1 (sect. 1.3) we noticed that \( E_1 \) can be implemented by means of an OR-gate with inputs \( y_0, y_1 \) and output \( y \). As a result \( E_0 \) can be implemented as, schematically, depicted in figure 2.2a, where \( B_1 \) (which stands for Bit Part 1) is specified by \( E_4 \). Notice that the two forks need not be isochronic since the decomposition is DI (cf. [JE1,2]).

To decompose \( E_4 \) we use the heuristic from [JE3] again:

Step 1: partitioning of outputs of \( E_4 \) : \( S.0 = \{ a_0^! \} \), \( S.1 = \{ b_0^! \} \), \( S.2 = \{ a_1^! \} \), \( S.3 = \{ b_1^! \} \);
Step 2: the corresponding input sets are: \( T.0 = \{ y_0^?, a^? \} \), \( T.1 = \{ y_0^?, b^? \} \), \( T.2 = \{ y_1^?, a^? \} \), and \( T.3 = \{ y_1^?, b^? \} \); Applying step 3, we define, for \( 0 \leq i \leq 4 \): \( E_4.i = E_4 \setminus (S.i \cup T.i) \), i.e.:

\[
E_4.0 = E_4 \setminus \{ y_0^?, a^?; a_0^! \} = \text{pref} \left[ (y_0^?; a^?; a_0^!)^2 \mid (y_0^?; b^?; b_0^!)^2 \mid (y_1^?; a^?; a_1^!)^2 \mid (y_1^?; b^?; b_1^!)^2 \right]
\]

\[
E_4.1 = E_4 \setminus \{ y_0^?, b^?; b_0^! \} = \text{pref} \left[ (y_0^?; b^?; b_0^!)^2 \mid (y_0^?; a^?; a_0^!)^2 \mid (y_1^?; a^?; a_1^!)^2 \mid (y_1^?; b^?; b_1^!)^2 \right]
\]

\[
E_4.2 = E_4 \setminus \{ y_1^?, a^?; a_1^! \} = \text{pref} \left[ (y_1^?; a^?; a_1^!)^2 \mid (y_1^?; b^?; b_1^!)^2 \mid (y_1^?; a^?; a_0^!)^2 \mid (y_1^?; b^?; b_0^!)^2 \right]
\]

\[
E_4.3 = E_4 \setminus \{ y_1^?, b^?; b_1^! \} = \text{pref} \left[ (y_1^?; b^?; b_1^!)^2 \mid (y_1^?; a^?; a_1^!)^2 \mid (y_1^?; b^?; b_0^!)^2 \mid (y_1^?; a^?; a_0^!)^2 \right]
\]

Notice (step 4) that \( (i: 0 \leq i \leq 4 : E_4.i) \) is a decomposition of \( E_4 \).
Now observing that $E_{4,i}$, with $0 \leq i < 4$, specifies a Muller C-element (the NCEL component in [JE1,p29]) leads to the implementation of BPI depicted in fig. 2.2b.

Generalization

The generalization of DMX1 to N-bits double rail coding, called DMXN, can easily be given. First notice that DMXN can be specified by:

$$EDRO = \text{pref} \left[ \left( i : 0 \leq i < N : \left( (y_0?, y!; a\cdot i?; a_{i,0}!)^2 \mid (y_0?, y!; b\cdot i?; b_{i,0}!)^2 \right) \mid \left( (y_1?, y!; a\cdot i?; a_{i,1}!)^2 \mid (y_1?, y!; b\cdot i?; b_{i,1}!)^2 \right) \right) \right]$$

where the data alphabet is $\{ i : 0 \leq i < N : a\cdot i, b\cdot i, a_{i,0}, a_{i,1}, b_{i,0}, b_{i,1} \}$. Here, $a\cdot i, b\cdot i$ is the coding of bit i of the sender, and $a_{i,j}, b_{i,j}$ is the coding of bit i of receiver j, with $0 \leq i < N \wedge 0 \leq j < 2$.

Following the same procedure as before, we find, as candidates for decomposition of EDRO:

$$EDR1 = \text{pref} \left[ (y_0?, y!)^2 \mid (y_1?, y!)^2 \right]$$

and, for $i : 0 \leq i < N$:

$$EDR_{4i} = \text{pref} \left[ (y_0?; a\cdot i?; a_{i,0}!)^2 \mid (y_0?; b\cdot i?; b_{i,0}!)^2 \mid (y_1?; a\cdot i?; a_{i,1}!)^2 \mid (y_1?; b\cdot i?; b_{i,1}!)^2 \right]$$

Notice that $EDR1$, $(i : 0 \leq i < N : EDR_{4i})$ is indeed a decomposition of EDRO. Notice also that $EDR1 = E1$, and that, for all i, $EDR_{4i}$ has the same form as $E4$. Since these components are DI-components, $EDR1$, $(i : 0 \leq i < N : EDR_{4i})$ is a DI-decomposition of EDRO.

To avoid global isochronic forks we therefore choose as cell structure: one cell containing the implementation of $EDR1$ (one 2-input-OR), and N cells containing the implementations of $EDR_{4i}, 0 \leq i < N$ (BPI: 4 C-elements).

Size and Delay

The size of the cell for $EDR1$ (one OR-gate) is $(3,2,6)$, the delay estimate is 3 (compare with ex. 1.4 and 1.5). The size of one C-element is $(4,4,12)$ (cf. ex. 1.6). Using two columns per input (and thus remaining height 4), the size of the cells for $EDR_{4i}$ is $(16,4,48)$. The delay of a C-element is 3 (cf. ex. 1.6). Since the C-elements in cells $EDR_{4i}$ are composed in parallel, the delay of cells $EDR_{4i}$ also is 3.

The size of the derived implementation for DMXN therefore is $N \cdot (16,4,48) + (3,2,6)$, and its delay, in both directions (request and data) is 3.
2.2.2 Solution 2

Derivation

As noticed in the derivation of solution 1, E1,E3 is a decomposition of E0. Instead of decomposing E3 into E4, gaining non-isochronicity of two forks, we now apply the heuristic of [JE3], used to find (i: 0\leq i<4: E4.i) in solution 1, directly to E3. We define, for 0\leq i<4: E3.i = E3 \upharpoonright (S.i \cup T.i). Notice that (i: 0\leq i<4: E3.i) is a decomposition of E3.

Let BP2 be the component specified by E3. The schematics of solution 2 and of BP2 are depicted in figures 2.3a and 2.3b respectively.

To derive implementations of components E3.i, 0\leq i<4, we consider the command of E3.0:

E3.0 = E3 \upharpoonright \{y_0?, a?, a_0!\}

= \text{pref} \{ (y_0?)^2 | (a?)^2 | (a_0!)^2 \}

The Karnaugh-map corresponding to E3.0 is given in figure 2.4a (where 'dc' stands for 'don't care'; no reachable state of E3.0 satisfies y_0 \land \neg a \land a_0).

\begin{align*}
\begin{array}{ccc|c}
\text{y_0} & 0 & 0 & 1 \\
\text{a_0} & 0 & 1 & 1 \\
0 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 & \text{dc}
\end{array}
\end{align*}

It is possible to fill in 1 at the don't care. Figure 2.4a then specifies a Muller C-element and in comparison to solution 1 we lost two non-isochronic forks. Hence, filling in 1 at dc is not a good suggestion. Filling in 0 at dc leads to figure 2.4b. This means a_0 can be specified by:

\begin{align*}
\begin{cases}
\neg a & \iff a_0 := 0 \\
\neg y_0 \land a & \iff a_0 := 1 \\
\neg y_0 \land a & \iff a_0 := a_0
\end{cases}
\end{align*}

Using the calculation rule derived in R5 (sect. 1.3), we find that this specification can be rewritten into:

a_0 := \neg x \quad \& \quad x := \neg(y_0 \land a) \land (\neg a \lor \neg a_0)

Noticing that the LHS of x's production equals \neg a \lor (\neg y_0 \land \neg a_0), which fortunately is a negative literal term, we get:
\[
\begin{align*}
& x \mapsto a_0 := 0 & a \land (y_0 \lor a_0) \mapsto x := 0 \\
& \neg x \mapsto a_0 := 1 & \neg a \lor (\neg y_0 \land \neg a_0) \mapsto x := 1
\end{align*}
\]

which can be translated directly into the circuit depicted in fig. 2.5; the asymmetric C-element from [CvB1].

**Size and Delay**

The implementation of E3.0 contains four gate nodes, each controlling 2 transistors. Its size therefore is (4,2,8). Only the internal gate node and the output node contribute to the delay estimate, with 1.5 (the resistance of path to 0 is 1.5 and the number of controlled n/p-pairs is 1), and 2 (the R-value is 1 ; the C-value is 2 since the number of controlled n/p-pairs is 1 and the additional capacitance for output nodes is 1 ) respectively. The delay estimate of E3.0 therefore is 3.5 .

To simplify the comparison with solution 1 we choose 4 as the height of the size estimate. As a result of mutual inputs the size of BP2 (4 asymmetric C-elements) is (12,4,32).

Since the asymmetric C-elements are composed in parallel, the delay estimate of BP2 is 3.5 .

![figure 2.5 : Implementation of E3.0.](image)

**Global isochronic forks**

In a similar way as in solution 1, the generalized demultiplexer DMXN can be realized with one OR-gate and N BP2-components. However, choosing a similar cell structure as in solution 1 causes global isochronic forks w.r.t. \( y_0 \) and \( y_1 \): see figure 2.3a. Since the isochronicity of the forks is asymmetric (i.e. it is essential that transitions of \( y_0,y_1 \) arrive at BP2 not later than they arrive at the OR-gate), these isochronic forks can be avoided by introducing internal variables, as will be demonstrated in the decomposition of E0 :

\[
\begin{align*}
E0 &= \text{pref} \left[ (y_0:y!;a?;a_0!)^2 \mid (y_0:y!;b?;b_0!)^2 \mid (y_1:y!;a?;a_1!)^2 \mid (y_1:y!;b?;b_1!)^2 \right] \\
D1 &\leftarrow \text{pref} \left[ (y_0:y!;z_0?;y!?;a?;a_0!)^2 \mid (y_0:y!;z_0?;y!?;b?;b_0!)^2 \mid (y_1:y!;z_1?;y!?;a?;a_1!)^2 \mid (y_1:y!;z_1?;y!?;b?;b_1!)^2 \right] \\
D1 &\leftarrow \text{pref} \left[ (z_0?;y!)^2 \mid (z_1?;y!)^2 \right] \\
\]
Let BP3 be the implementation of the last component of the decomposition. We then conclude that DMXI can be implemented as depicted in figure 2.6a. Now observe that:

\[
\text{pref} \left[ (y_0?:z_0!:a?:a_0!)^2 \mid (y_0?:z_0!:b?:b_0!)^2 \mid (y_1?:z_1!:a?:a_1!)^2 \mid (y_1?:z_1!:b?:b_1!)^2 \right] \\
\quad \text{pref} \left[ y_0?:z_0! \right] \\
\quad , \text{pref} \left[ y_1?:z_1! \right] \\
\quad , \text{pref} \left[ (z_0?:a?:a_0!)^2 \mid (z_0?:b?:b_0!)^2 \mid (z_1?:a?:a_1!)^2 \mid (z_1?:b?:b_1!)^2 \right]
\]

Observing that the first two components of this decomposition can be implemented by WIRE-components (see [JE1]), and that the latter component can be implemented by a BP2-component, we conclude that BP3 can be implemented as depicted in figure 2.6b.

**Generalization**

Similar as in solution 1, DMXN can be implemented with 1 OR-gate and N BP3-components. Choosing a similar cell structure as in solution 1 we get: one cell containing a 2-input-OR and N cells containing a BP3-component (see fig. 2.6c). The costs of the implementation of BP3 equal the costs of the implementation of BP2. Therefore, the size of this implementation of DMXN is \( N \cdot (12,4,32) + (3,2,6) \). The delay in the request direction is 3 (cf. sol. 1), and the delay in the data direction is 3.5.

**Generalization**

Similar as in solution 1, DMXN can be implemented with 1 OR-gate and N BP3-components. Choosing a similar cell structure as in solution 1 we get: one cell containing a 2-input-OR and N cells containing a BP3-component (see fig. 2.6c). The costs of the implementation of BP3 equal the costs of the implementation of BP2. Therefore, the size of this implementation of DMXN is \( N \cdot (12,4,32) + (3,2,6) \). The delay in the request direction is 3 (cf. sol. 1), and the delay in the data direction is 3.5.
2.2.3 Solution 3

Derivation

To derive the third implementation of DMX1 we restart with specification EO:

\[ EO = \text{pref} [(y_0?; y!; a?; a_0!)^2 \mid (y_0?; y!; b?; b_0!)^2 \mid (y_1?; y!; a?; a_1!)^2 \mid (y_1?; y!; b?; b_1!)^2] \]

In the previous sections we noticed:

i) the data outputs, \( a_0!, a_1!, b_0!, b_1! \), do not depend on the data inputs, \( a?, b? \), only.

ii) to determine outputs \( a_0, a_1, b_0, b_1 \) from inputs \( y_0, y_1, a, b \) it is necessary to consider additional information about the current state, e.g. in the first alternative of EO as well as in the third alternative of EO there are states satisfying \( \neg y_0 \land \neg y_1 \land a \land \neg b \), they satisfy either \( a_0 \land \neg a_1 \) (in the first alternative) or \( \neg a_0 \land a_1 \) (in the third alternative). In other words: to determine the data outputs a memory function is required.

iii) BP, the part of DMX1 that produces the data outputs, is required \( N \) times in the generalization of DMX to \( N \) bits double rail coding, whereas RP, the part of DMX1 that produces the request outputs, is required only once in this generalization.

In solution 1 and 2 the memory function mentioned in ii) is implemented in BP by means of a feedback loop of the data outputs.

On account of iii) it is advantageous to implement BP as cheap as possible, even if this means that RP has to be implemented more expensively.

In solution 3 we shall therefore introduce internal variables such that:

- they realize the memory function mentioned in ii.
- they are produced in RP.

From i and ii) we conclude that it is helpful to be able to distinguish between the first two alternatives of EO on the one hand and the latter two alternatives on the other hand. Therefore we introduce, as a first step towards introducing appropriate internal variables, thinkvariables \( t_0 \) and \( t_1 \) specified by:

for \( i, 0 \leq i < 2 \) : \( t_i \equiv \text{'receiver } i \text{' is involved in the transport protocol}' \)

Within EO this can be translated into:

initially \( \neg t_0 \land \neg t_1 \\
EO' = \text{pref} [ (y_0? \circ t_0\uparrow); y!; a?; a_0!; y_0?; y!; a?; (a_0! \circ t_0\downarrow) \mid (y_0? \circ t_0\uparrow); y!; b?; b_0!; y_0?; y!; b?; (b_0! \circ t_0\downarrow) \mid (y_1? \circ t_1\uparrow); y!; a?; a_1!; y_1?; y!; a?; (a_1! \circ t_1\downarrow) \mid (y_1? \circ t_1\uparrow); y!; b?; b_1!; y_1?; y!; b?; (b_1! \circ t_1\downarrow) ] \]
A command part between angle brackets constitutes an atomic action, i.e. in \( \langle \text{act}_0 \diamond \text{act}_1 \rangle \) the executions of actions \( \text{act}_0 \) and \( \text{act}_1 \) are defined to happen simultaneously. Since the role of \( t_0 \) and \( t_1 \) is merely one of thinkvariables it is allowed to consider the parts between angle brackets in \( EO' \) as atomic.

We now observe that the data outputs can be specified by:

\[(s1) \text{ for } i, 0 \leq i < 2: \quad a_i := a \land t_i \quad \& \quad b_i := b \land t_i \]

This suggests to try introducing internal variables \( x_0 \) and \( x_1 \) such that the data outputs can be specified by:

\[(s2) \text{ for } i, 0 \leq i < 2: \quad a_i := a \land x_i \quad \& \quad b_i := b \land x_i \]

or, similarly, introducing \( x_0 \) and \( x_1 \) such that the data outputs can be specified by:

\[(s3) \text{ for } i, 0 \leq i < 2: \quad a_i := a \land \neg x_i \quad \& \quad b_i := b \land \neg x_i \]

Specification \((s2)\) leads to 4 AND-gates as implementation of \( \text{BP} \). As will be shown, specification \((s3)\) leads to 4 NOR-gates and 2 Inverters as implementation of \( \text{BP} \). The latter implementation is cheaper w.r.t. size of \( \text{BP} \): \((8,4,24)\) versus \((6,4,20)\). Since our aim is to implement \( \text{BP} \) as cheap as possible we continue with specification \((s3)\).

From \((s1)\) we derive requirements on \( x_0 \) and \( x_1 \) in order to establish \((s3)\):

- It is necessary that \( a \land \neg x_1 \) and \( b \land \neg x_1 \), \( 0 \leq i < 2 \), do not hold unjustly, which means:

  \[
  \text{for } i, 0 \leq i < 2: \quad \neg (a \lor b) \Rightarrow x_i
  \]

  On account of \( a \lor b \Rightarrow t_0 \neq t_1 \) this equivales:

  \[
  (t_0 \land (a \lor b) \Rightarrow x_1) \land (t_1 \land (a \lor b) \Rightarrow x_0)
  \]

  For the first alternative of \( EO' \), i.e. \( \langle y_0? \diamond t_0\uparrow \rangle; y!; a?; a_0!; y_0?; y!; a?; \langle a_0? \diamond t_0\downarrow \rangle \), this means that \( x_1 \) must hold before \( a\uparrow \). Since \( a\uparrow \) is an input, this implies that \( x_1 \) must be a postcondition of \( y\uparrow \). As a result, \( x_1 \) must be a precondition of \( y\uparrow \). Since \( x_1 \) is a postcondition of \( \tilde{x}_1\uparrow \), this can be established by letting \( \tilde{x}_1\uparrow \) precede \( y\uparrow \).

- It is necessary that \( a \land \neg x_i \) and \( b \land \neg x_i \), \( 0 \leq i < 2 \), do hold if necessary according to \((s3)\), which means:

  \[
  \text{for } i, 0 \leq i < 2: \quad t_i \land (a \lor b) \Rightarrow \neg x_i
  \]

  This means that, in case of \( t_1 \), \( \neg x_1 \) is a precondition of \( a_i\uparrow \). Similar to the above this can be established by letting \( \tilde{x}_i\downarrow \) precede \( a_i\uparrow \).

Using this, we derive:

\[
\text{ES'} = \preff\{ \langle y_0? \diamond t_0\uparrow \rangle; \tilde{x}_0\downarrow; (\tilde{x}_1\uparrow; y!; a?) \; a_0!; y_0?; y!; a?; \langle a_0! \diamond t_0\downarrow \rangle \}
\mid \langle y_0? \diamond t_0\uparrow \rangle; \tilde{x}_0\downarrow; (\tilde{x}_1\uparrow; y!; b?) \; b_0!; y_0?; y!; b?; \langle b_0! \diamond t_0\downarrow \rangle
\mid \langle y_1? \diamond t_1\uparrow \rangle; \tilde{x}_1\downarrow; (\tilde{x}_0\uparrow; y!; a?) \; a_1!; y_0?; y!; a?; \langle a_1! \diamond t_1\downarrow \rangle
\mid \langle y_1? \diamond t_1\uparrow \rangle; \tilde{x}_1\downarrow; (\tilde{x}_0\uparrow; y!; b?) \; b_1!; y_0?; y!; b?; \langle b_1! \diamond t_1\downarrow \rangle
\}
in which the data outputs are specified by (s3). Since the thinkvariables are no longer needed we omit them:

$$E_5 = \text{pref } [ y_0?; x_0\downarrow \| (x_1\uparrow; y!; a?); a_0!/ y_0?; y!; a?; a_0!$$

$$| y_0?; x_0\downarrow \| (x_1\uparrow; y!; b?); b_0!/ y_0?; y!; b?; b_0!$$

$$| y_1?; x_1\downarrow \| (\bar{x}_0\uparrow; y!; a?); a_1!/ y_1?; y!; a?; a_1!$$

$$| y_1?; x_1\downarrow \| (\bar{x}_0\uparrow; y!; b?); b_1!/ y_1?; y!; b?; b_1!$$

Now notice that $E_5 \cup E_0 = E_0$, i.e., $E_5$ is an expansion of $E_0$ ([JE3,p9]). So we now have accomplished step 5 of the heuristic and can apply steps 1 to 4. With step 1 and 2 we find, as candidates for a decomposition of $E_0$:

$$E_6 = E_5 \uparrow \{ y_0?, y_1?, x_0!, x_1! \}$$

$$= \text{pref } [ y_0?; x_0\downarrow \| (x_1\uparrow; y!; y!); y_0?; y!; y_0?; y! ]$$

$$E_7 = E_5 \uparrow \{ y_0?, y_1?, x_0!, x_1!, y! \}$$

$$= \text{pref } [ y_0?; x_0\downarrow \| (x_1\uparrow; y!); y_0?; y!; y_1?; \bar{x}_1\downarrow \| (\bar{x}_0\uparrow; y!); y_1?; y! ]$$

$$E_8 = E_5 \uparrow \{ x_0?, x_1?, a?, b?, a_0?, b_0?, a_1?, b_1! \}$$

$$= \text{pref } [ \bar{x}_0\downarrow \| (x_1\uparrow; a?); a_0!/ a_0?; a_0! \| \bar{x}_0\downarrow \| (x_1\uparrow; b?); b_0!/ b_0?; b_0!$$

$$| \bar{x}_1\downarrow \| (\bar{x}_0\uparrow; a?); a_1!/ a_1?; a_1! \| \bar{x}_1\downarrow \| (\bar{x}_0\uparrow; b?); b_1!/ b_1?; b_1!$$

It can be verified, using the theory from [JE1,2,3], that $E_6$, $E_7$, $E_8$ is a decomposition of $E_0$.

Let RP1, RP2, and BP4 be the components specified by $E_6$, $E_7$, $E_8$ respectively.

**Implementations**

The derivation of implementations of RP1 and RP2 can be done rather straightforwardly using p.r. sets. In the derivations below the first p.r. sets are derived immediately from the directed commands $E_6$ and $E_7$, using the additional information about the initial state: $\neg y_0 \land \neg y_1 \land \neg y$. In the first steps of the derivations we make convenient choices regarding 'dc'. For RP1, the derivation leads to an implementation with cross coupled NOR-gates: fig. 2.7a. The derivation for RP2 needs an additional step that introduces an internal variable to derive a complementary MOS implementation (cf. CR3 in sect. 1.3). It leads to the implementation of RP2 depicted in fig. 2.7b.
Derivation for RP1:

\[
\begin{align*}
&\{ y_0 \land y_1 \implies dc, \\
& y_0 \land \neg y_1 \implies x_0 := 0 \land x_1 := 1, \\
& \neg y_0 \land y_1 \implies x_0 := 0 \land x_1 := 0, \\
& \neg y_0 \land \neg y_1 \implies x_0 := x_0 \land x_1 := x_1, \\
& x_0 := 0 \land x_1 := 0, \\
& x_0 := 0 \land x_1 := x_0, \\
& x_0 := x_0 \land x_1 := x_1, \\
& x_0 := 0 \land x_1 := x_0, \\
& x_0 := x_0 \land x_1 := x_1 \}
\end{align*}
\]

Derivation for RP2:

\[
\begin{align*}
&\{ y_0 \land y_1 \implies dc, \\
& y_0 \land \neg y_1 \implies y := x_1, \\
& \neg y_0 \land y_1 \implies y := x_0, \\
& \neg y_0 \land \neg y_1 \implies y := 0, \\
& y := (x_0 \lor x_1), \\
& y := x_1, \\
& y := x_0, \\
& y := 0, \\
& (x_0 \land x_1) \lor (y_1 \land x_0) \implies y := 1, \\
& (\neg y_0 \land \neg x_1) \land (\neg y_1 \land \neg x_0) \implies y := 0, \\
& (x_0 \land x_1) \lor (y_1 \land x_0) \implies z := 0, \\
& (\neg y_0 \land \neg x_1) \land (\neg y_1 \land \neg x_0) \implies z := 1, \\
& z := y := 0, \\
& \neg z := y := 1
\end{align*}
\]

figure 2.7a: Implementation of RP1. figure 2.7b: Implementation of RP2.
Finding an implementation of BP4 is more complicated. Applying steps 1 and 2 of the heuristic (sect. 1.2) we find the following sets: \( R_1 = \{ x_0, a, a_0 \} \), \( R_2 = \{ x_0, b, b_0 \} \), \( R_3 = \{ x_1, a, a_1 \} \), and \( R_4 = \{ x_1, b, b_1 \} \). Step 3 gives the commands: \( E_8.i = E_8 \uparrow R_i \), which form a decomposition of \( E_8 \). The commands \( E_8.i \) have the form:

\[
E_8.0 = \text{pref} [\overline{x}_0 \downarrow \uparrow; a?; a_0\lnot; a?; a_0!; \mid \overline{x}_0 \downarrow \uparrow; a?; a_0!; \mid \overline{x}_0 \uparrow \downarrow; a?; a_0\lnot; a?; a_0!]
\]

The p.r. set for \( E_8.0 \) is:

- \( \neg a \lor x_0 \longmapsto a_0 := 0 \)
- \( a \land \neg x_0 \longmapsto a_0 := 1 \)

This p.r. set cannot be rewritten such that the guards of the productions are positive and negative literal terms. As a result it is difficult to find a complementary MOS implementation using this p.r. set. We therefore introduce two internal variables in \( E_8 \) in the following way:

\[
E_8 = \text{pref} [\overline{x}_0 \downarrow \uparrow; a?; a_0\lnot; a?; a_0!; \mid \overline{x}_0 \downarrow \uparrow; a?; a_0!; \mid \overline{x}_0 \downarrow \uparrow; a?; a_0\lnot; a?; a_0!]
\]

Now, the candidates for decomposition found by applying the heuristic are:

- \( \text{pref} [a?; p!] \), \( \text{pref} [b?; q!] \)
- \( \text{pref} [\overline{x}_0 \downarrow \uparrow; a?; a_0\lnot; a?; a_0!; \mid \overline{x}_0 \uparrow \downarrow; p?; p?; \mid x_0 \] \)
- \( \text{pref} [\overline{x}_0 \downarrow \uparrow; q?; b_0\lnot; q?; b_0!; \mid \overline{x}_0 \uparrow \downarrow; q?; q?; \mid x_0 \] \)
- \( \text{pref} [\overline{x}_1 \downarrow \uparrow; a?; a_0\lnot; a?; a_0!; \mid \overline{x}_1 \uparrow \downarrow; q?; q?; \mid x_1 \] \)
- \( \text{pref} [\overline{x}_1 \downarrow \uparrow; q?; b_0\lnot; q?; b_0!; \mid \overline{x}_1 \uparrow \downarrow; q?; q?; \mid x_1 \] \)

Notice that they form a decomposition of \( E_8 \). Furthermore, notice that the latter components differ from the components \( E_8.i \), since the initial valuation of \( p \) and \( q \) is \( p \land q \). As a result of this, the initial valuation of the first two components of this decomposition can be implemented by Inverters. The other 4 components are equally formed, so it suffices to consider one of them. The p.r. set of \( \text{pref} [\overline{x}_0 \downarrow \uparrow; a?; a_0\lnot; a?; a_0!; \mid \overline{x}_0 \uparrow \downarrow; p?; p?; \mid x_0 \] \) is:

- \( \neg p \land \neg x_0 \longmapsto a_0 := 0 \)
- \( p \lor x_0 \longmapsto a_0 := 1 \)
This p.r. set specifies a NOR-gate. The thus found implementation of BP4 is depicted in figure 2.8.

![Diagram of BP4 implementation](image)

**figure 2.8 : Implementation of BP4.**

**Global isochronic forks**

Similar as in the previous solutions DMXN, the generalization of DMX1, can be implemented by one request part, formed by RP1 and RP2, and N data parts containing a BP4-component. As in solution 2 global isochronic forks exist between the request part and the data parts. These global isochronic forks can be avoided in the same way as in solution 2:

\[
\begin{align*}
E_0 &= \text{pref} \left[ (y_0; y!; a?; a_0!)^2 \mid (y_0; y!; b?; b_0!)^2 \mid (y_1; y!; a?; a_1!)^2 \mid (y_1; y!; b?; b_1!)^2 \right] \\
D_1 \rightarrow \text{pref} \left[ y_0; \bar{x}_0 \downarrow \right] \left( \bar{x}_0 \uparrow; y!; a? \right) \left( a_0!; y_0?; y!; a?; a_0! \right) \\
&\quad \left( y_0; \bar{x}_0 \downarrow \right) \left( \bar{x}_1 \uparrow; y!; b? \right) \left( b_0!; y_0?; y!; b?; b_0! \right) \\
&\quad \left( y_1; \bar{x}_1 \downarrow \right) \left( \bar{x}_0 \uparrow; y!; a? \right) \left( a_1!; y_1?; y!; a?; a_1! \right) \\
&\quad \left( y_1; \bar{x}_1 \downarrow \right) \left( \bar{x}_0 \uparrow; y!; b? \right) \left( b_1!; y_1?; y!; b?; b_1! \right) \\
D_1 \rightarrow \text{pref} \left[ y_0; \left( \bar{x}_0 \downarrow; \bar{v}_0 \downarrow \right) \left( \bar{x}_1 \uparrow; \bar{v}_1 \uparrow; y! \right) \left( a_0!; y_0?; y!; a?; a_0! \right) \\
&\quad \left( y_0; \left( \bar{x}_0 \downarrow; \bar{v}_0 \downarrow \right) \left( \bar{x}_1 \uparrow; \bar{v}_1 \uparrow; y! \right) \left( b_0!; y_0?; y!; b?; b_0! \right) \\
&\quad \left( y_1; \left( \bar{x}_1 \downarrow; \bar{v}_1 \downarrow \right) \left( \bar{x}_0 \uparrow; \bar{v}_0 \uparrow; y! \right) \left( a_1!; y_1?; y!; a?; a_1! \right) \\
&\quad \left( y_1; \left( \bar{x}_1 \downarrow; \bar{v}_1 \downarrow \right) \left( \bar{x}_0 \uparrow; \bar{v}_0 \uparrow; y! \right) \left( b_1!; y_1?; y!; b?; b_1! \right) \\
D_1 \rightarrow \text{pref} \left[ y_0; \left( \bar{x}_0 \downarrow; \bar{v}_0 \downarrow \right) \left( \bar{x}_1 \uparrow; \bar{v}_1 \uparrow; y! \right) \right] \\
&\quad \left( y_1; \left( \bar{x}_1 \downarrow; \bar{v}_1 \downarrow \right) \left( \bar{x}_0 \uparrow; \bar{v}_0 \uparrow; y! \right) \right] \\
&\quad \text{pref} \left[ \left( \bar{x}_0 \downarrow; \bar{v}_0 \downarrow \right) \left( \bar{x}_1 \uparrow; \bar{v}_1 \uparrow \right) \left( a? \right) \left( a_0!; a?; a_0! \right) \\
&\quad \left( \bar{x}_0 \downarrow; \bar{v}_0 \downarrow \right) \left( \bar{x}_1 \uparrow; \bar{v}_1 \uparrow \right) \left( b? \right) \left( b_0!; b?; b_0! \right) \\
&\quad \left( \bar{x}_1 \downarrow; \bar{v}_1 \downarrow \right) \left( \bar{x}_0 \uparrow; \bar{v}_0 \uparrow \right) \left( a? \right) \left( a_1!; a?; a_1! \right) \\
&\quad \left( \bar{x}_1 \downarrow; \bar{v}_1 \downarrow \right) \left( \bar{x}_0 \uparrow; \bar{v}_0 \uparrow \right) \left( b? \right) \left( b_1!; b?; b_1! \right) \\
\right]
\]
Let RP3 and BPS be the components specified by the last two directed commands of the decomposition above. Now notice that RP3 can be decomposed into:

\[
\begin{align*}
&\text{pref } \langle y_0; x_0 \uparrow; x_1 \uparrow; y_1 \uparrow \rangle, \\
&\text{pref } \langle y_0; \overline{x}_0 \downarrow; (\overline{y}_1 \uparrow; y \uparrow) ; y_0 \downarrow; y_1 \downarrow \rangle\text{ (over)};
\end{align*}
\]

Since these are the specifications of RP1 and RP2 (with \(v_0, v_1\) instead of \(x_0, x_1\)), RP3 can be implemented by one RP1-component and one RP2-component.

Notice that BPS can be decomposed into:

\[
\begin{align*}
&\text{pref } \langle x_0 \uparrow; \overline{v}_0 \downarrow; (\overline{x}_0 \uparrow; \overline{v}_1 \downarrow) \rangle, \\
&\text{pref } \langle x_0 \uparrow \rangle, \text{ pref } \langle X_1 \uparrow; \overline{V}_1 \downarrow \rangle\text{, E8 .}
\end{align*}
\]

This equals: \(\text{pref } \langle x_0 \uparrow; \overline{v}_0 \downarrow \rangle, \text{ pref } \langle x_1 \uparrow; \overline{v}_1 \downarrow \rangle\text{, E8} .\) As a result, BPS can be implemented by two WIRE-components and one BP4-component.

The thus found implementation of DMX1 is depicted in figure 2.9a.

**Generalization**

In a similar way as solution 2, solution 3 for DMX1 can be generalized to DMXN. This generalization is depicted in fig. 2.9b.

**Size**

RP3 (see fig.2.7a,b and 2.9a) contains seven gate nodes (viz. \(y_0, y_1, x_0, x_1, v_0, v_1\), and one internal node in RP2), so the width of the cell area is 7. Two of these gate nodes control four transistors, the other gate nodes control two transistors, so the height of the cell area is 4. Since the number of transistors is 18, the size of RP3 is (7,4,18). Notice that it is possible to use two columns for each of the gate nodes that control four transistors (see remark 1 sect.1.4), which leads to size (9,2,18) for RP3.
Component BP5 (fig. 2.8 and 2.9a) contains six gate nodes (viz. $a$, $b$, $p$, $q$, $x_0$, $x_1$). The nodes $p$, $q$, $x_0$, and $x_1$ control four transistors. Since the total number of transistors in BP5 is 20, the size of BP5 is $(6,4,20)$.

The size of solution 3 for DMXN therefore is $N \cdot (6,4,20) + (7,4,18)$.

**Delay**

The outputs of the cross coupled NOR-gates in RP1 (fig. 2.7a) control an additional n/p-pair, as a result of which the delay estimate of each NOR-gate is 3 (instead of 2 as in ex. 1.4). Since the NOR-gates are cross coupled the total delay estimate of RP1 is 6.

The R-value of the internal gate node in RP2 (see fig. 2.7b) is 2, since in case of (dis-)charging of this node only one path (with resistance 2) to a source is conducting. Since this node controls one n/p-pair, its C-value is 1. The contribution to the delay estimate of the internal gate node therefore is 2. The contribution to the delay estimate of $y$ is 1, and consequently the delay estimate of RP1 is 3.

Since these components are composed in series in RP3 ($v_3$ has the same delay as $x_3$), the delay estimate of RP3 is 9.

In BP5 (see fig. 2.8 and 2.9a) the output of each Inverter controls two n/p-pairs, as a result of which the contribution to the delay estimate of the Inverters is 2 (compare with ex. 1.3c). The delay of a NOR-gate is 2 (ex. 1.4). The delay of BP5 therefore is 4.

**Conclusion**: The delay of solution 3 for DMXN in the request direction is 9, and its delay in the data direction is 4.

### 2.3 Comparison

In table 2.1 below the size and delays of solutions 1, 2, and 3 are given. In this table $N$ is the number of bits, and 'r.(d.)-delay' stands for 'delay in the request (data) direction'.

Notice that the delays do not depend on the number of bits. Solution 3 has the smallest size, but also—in both directions—the largest delay.

<table>
<thead>
<tr>
<th>solution nr.</th>
<th>size</th>
<th>r.-delay</th>
<th>d.-delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$N \cdot (16,4,48) + (3,2,6)$</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>$N \cdot (12,4,32) + (3,2,6)$</td>
<td>3</td>
<td>3.5</td>
</tr>
<tr>
<td>3</td>
<td>$N \cdot (6,4,20) + (7,4,18)$</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

table 2.1: Costs of solutions 1, 2, and 3.
3 Concluding Remarks

Starting with a component specified by a directed command satisfying conditions A1-3 (given in sect. 1.2), our design path contains the following steps. Using the heuristic given in sect. 1.2 and the general design strategy w.r.t. cell-structure (sect. 1.1), we (DI-) decompose (using the theory from [JE1,2,3]) the directed command into a network of (smaller) directed commands also satisfying A1-3. Once the found components are small enough, we use production rule sets to find (C) MOS implementations for these components. To compare different components the cost measures given in sect. 1.4 are used.

The heuristic given in sect. 1.2 gives a good lead for finding decompositions. However, this heuristic does not tell exactly how to introduce internal variables. In the derivation presented in section 2.2.3 we successfully used ghost variables to introduce internal variables, and we demonstrated the usefulness of conditional transitions (sect. 1.2).

Condition A3 on directed commands, the notion conditional transition, as well as the p.r. sets are directed towards the state-based arguments that are convenient at transistor level rather than towards the transition-based arguments used at higher levels.

Production rule sets (p.r. sets) are a convenient tool for deriving CMOS implementations for (small) components. To complete the theory for p.r. sets, correctness proofs for the translation to MOS circuits and for the calculation rules are needed. The given calculation rules do not suffice for practical purposes (e.g., in the derivation of an implementation for RP1 (sect. 2.2.3), a new calculation rule is used). Although working towards the form of p.r. sets described in R3 (sect. 1.3) gives a good guideline, a heuristic for calculating with p.r. sets (including the use of the freedom of implementation (filling in 'dc')) is needed. To overcome the disadvantage mentioned in the concluding remarks in section 1.3 (viz. some implementations can not be represented by p.r. sets), the form of p.r. sets or the translation to transistor networks could be changed. Such a change would, however, complicate the correctness proofs and the calculation rules.

In chapter 2 the methods described in chapter 1 are used to derive a number of implementations of the demultiplexer. As usual, smaller size for implementations coincides with larger delay (cf. table 2.1, sect. 2.3).
Appendix A  Proof of theorem 1

In this appendix the proof of theorem 1 (section 1.3) is given. Throughout the appendix, 
\( x \) denotes an arbitrary element of \( X \), and \( f, g, \) and \( h \) denote arbitrary boolean terms.

Definition

Let \( t, t \in BT \rightarrow BT \) be defined by :

\[
\begin{align*}
D0 & : t(x) = x \\
D1 & : t(\neg x) = \neg x \\
D2 & : t(f \lor g) = t(f) \lor t(g) \\
D3 & : t(f \land g) = t(f) \land t(g) \\
D4 & : t(\neg f) = t(f) \\
D5 & : t(\neg (f \lor g)) = t(\neg f) \land t(\neg g) \\
D6 & : t(\neg (f \land g)) = t(\neg f) \lor t(\neg g)
\end{align*}
\]

Notice that \( f \) can be transformed into \( t(f) \) by applying the transformation rules given in theorem 1.

Let the length of a boolean term be defined as the number of occurrences of elements of \( X \cup \{\neg, \land, \lor\} \), i.e. the length of the string, not counting brackets.

The following two properties can easily be proved with induction to the length of \( f \).

\( P0 \quad f \in BT \Rightarrow (t(f) \in LT \land f = t(f)) \)

\( P1 \quad f \in LT \Rightarrow t(f) \equiv f \)

Corollary

\( C0 \quad t \) is idempotent, i.e. \( f \in BT \Rightarrow t(t(f)) \equiv t(f) \)

The main theorem (theorem 1), viz. the order of applications of the transformation rules is indifferent, can now be formalized :

Theorem

\[
\begin{align*}
T0 & : t(f) = t(t(f)) \\
T1 & : t(\neg f) = t(\neg t(f)) \\
T2 & : t(f \lor g) = t(t(f) \lor t(g)) \\
& \quad = t(f \lor t(g)) \\
& \quad = t(t(f) \lor g) \\
T3 & : t(f \land g) = t(f) \land t(g) \\
& \quad = t(t(f) \land t(g)) \\
& \quad = t(t(f) \land g)
\end{align*}
\]
proof:

The proofs of \( T_0 \), \( T_2 \), and \( T_3 \) are easy using \( C_0 \), \( D_2 \), and \( D_3 \). For instance, the first part of \( T_2 \):

\[
\begin{align*}
t(t(f) & \lor t(g)) \\
& \equiv \{ D_2 \} \\
t(t(f)) & \lor t(t(g)) \\
& \equiv \{ C_0 \} \\
t(f) & \lor t(g) \\
& \equiv \{ D_2 \} \\
t(f \lor g)
\end{align*}
\]

\( T_1 \) is proved with induction to the length of the boolean term involved, where the length of \( f \) is denoted by \( l(f) \):

The base of the induction, \( f \equiv x \) and \( f \equiv \neg x \), is easily verified using \( D_0 \) and \( D_1 \).

In the induction step we consider \( f \) having the form \( g \lor h \), \( g \land h \), \( \neg(g \lor h) \), \( \neg(g \land h) \), or \( \neg g \). In the proofs 'induction hypothesis' is abbreviated as 'i.h.'.

- If \( f \equiv g \lor h \), then:

\[
\begin{align*}
t(-t(f)) & \\
& \equiv [ f \equiv g \lor h ] \\
t(-t(g \lor h)) & \\
& \equiv \{ D_2 \} \\
t(-t(g) \lor t(h))) & \\
& \equiv \{ D_5 \} \\
t(-t(g)) & \land t(-t(h)) \\
& \equiv \{ i.h., \text{using } l(g) < l(g \lor h) \text{ and } l(h) < l(g \lor h) \} \\
t(-g) & \land t(-h) \\
& \equiv \{ D_5 \} \\
t(-(g \lor h)) & \\
& \equiv [ f \equiv g \lor h ] \\
t(-f)
\end{align*}
\]

- If \( f \equiv g \land h \), the proof is similar to the one above.

- If \( f \equiv \neg(g \lor h) \), then:

\[
\begin{align*}
t(-t(f)) & \\
& \equiv [ f \equiv \neg(g \lor h) ] \\
t(-t(\neg(g \lor h))) & \\
& \equiv \{ D_5 \} \\
t(\neg(-t(g) \land t(-h)))
\end{align*}
\]
\[ \begin{align*}
&= \text{(D6)} \\
&= \tau(\neg \neg (\neg \neg \neg \neg)) \\
&\lor \tau(\neg \neg (\neg \neg \neg \neg)) \\
&= \{ \text{i.h., using } l(\neg g) < l(\neg (g \lor h)) \text{ and } l(\neg h) < l(\neg (g \lor h)) \} \\
&= \tau(\neg g) \lor \tau(\neg h) \\
&= \text{(D4)} \\
&= \tau(g) \lor \tau(h) \\
&= \text{(D2)} \\
&= \tau(g \lor h) \\
&= \text{(D4)} \\
&= \tau(\neg (g \lor h)) \\
&\Rightarrow \{ f \equiv \neg (g \lor h) \} \\
&= \tau(\neg f) \\
&\Rightarrow \text{If } f \equiv \neg (g \land h), \text{ the proof is similar to the one above.} \\
&\Rightarrow \text{If } f \equiv \neg \neg g, \text{ then:} \\
&\Rightarrow \tau(\neg \neg \neg \neg) \\
&\Rightarrow \{ f \equiv \neg \neg g \} \\
&\Rightarrow \tau(\neg \neg \neg \neg) \\
&\Rightarrow \text{(D4)} \\
&\Rightarrow \tau(\neg \neg \neg \neg) \\
&\Rightarrow \{ \text{i.h., using } l(g) < l(\neg \neg \neg \neg) \} \\
&\Rightarrow \tau(\neg g) \\
&\Rightarrow \text{(D4)} \\
&\Rightarrow \tau(\neg \neg \neg \neg) \\
&\Rightarrow \{ f \equiv \neg \neg g \} \\
&\Rightarrow \tau(\neg f)
\end{align*} \]

\[\Box\]

The following property can easily be proved with induction to the length of \( f \):

\[\text{P2 } \tau(f) \in NLT \Rightarrow \tau(\neg f) \in PLT\]

Using \( \tau(f) \equiv \text{lt}(f) \), (2) follows from P0, (3) follows from P1, and (4) follows from P2.
Acknowledgements

Acknowledgements are due to Kees van Berkel for explaining the notions regarding cost measures that are formalized in section 1.4, and to Wim Kloosterhuis who came up with the first ideas that eventually led to solution 3 (section 2.2.3). Thanks are also due to Tom Verhoeff and Ad Peeters for their critical remarks on earlier drafts of this paper.

References


In this series appeared:

91/01 D. Alstein

91/02 R.P. Nederpelt H.C.M. de Swart
Implication. A survey of the different logical analyses “if...,then...”, p. 26.

91/03 J.P. Katoen L.A.M. Schoenmakers
Parallel Programs for the Recognition of P-invariant Segments, p. 16.

Performance Analysis of VLSI Programs, p. 31.

91/05 D. de Reus
An Implementation Model for GOOD, p. 18.

91/06 K.M. van Hee
SPECIFICATIEMETHODEN, een overzicht, p. 20.

91/07 E. Poll
CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.

91/08 H. Schepers
Terminology and Paradigms for Fault Tolerance, p. 25.

91/09 W.M.P.v.d.Aalst
Interval Timed Petri Nets and their analysis, p. 53.

91/10 R.C. Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude
POLYNOMIAL RELATORS, p. 52.

91/11 R.C. Backhouse P.J. de Bruin G. Malcolm E. Voermans J. van der Woude
Relational Catamorphism, p. 31.

91/12 E. van der Sluis

91/13 F. Rietman
A note on Extensionality, p. 21.

91/14 P. Lemmens
The PDB Hypermedia Package. Why and how it was built, p. 63.

91/15 A.T.M. Aerts K.M. van Hee

91/16 A.J.J.M. Marcelis
An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.

91/17 A.T.M. Aerts P.M.E. de Bra K.M. van Hee
Transforming Functional Database Schemes to Relational Representations, p. 21.
<table>
<thead>
<tr>
<th>Paper Number</th>
<th>Author(s)</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>91/18</td>
<td>Rik van Geldrop</td>
<td>Transformational Query Solving, p. 35.</td>
</tr>
<tr>
<td>91/19</td>
<td>Erik Poll</td>
<td>Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.</td>
</tr>
<tr>
<td>91/23</td>
<td>K.M. van Hee, L.J. Somers, M. Voorhoeve</td>
<td>Z and high level Petri nets, p. 16.</td>
</tr>
<tr>
<td>91/24</td>
<td>A.T.M. Aerts, D. de Reus</td>
<td>Formal semantics for BRM with examples, p. 25.</td>
</tr>
<tr>
<td>91/25</td>
<td>P. Zhou, J. Hoorman, R. Kuiper</td>
<td>A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and completeness, p. 52.</td>
</tr>
<tr>
<td>91/27</td>
<td>F. de Boer, C. Palamidessi</td>
<td>Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.</td>
</tr>
<tr>
<td>91/28</td>
<td>F. de Boer</td>
<td>A compositional proof system for dynamic process creation, p. 24.</td>
</tr>
<tr>
<td>91/30</td>
<td>J.C.M. Baeten, F.W. Vaandrager</td>
<td>An Algebra for Process Creation, p. 29.</td>
</tr>
<tr>
<td>91/31</td>
<td>H. ten Eikelder</td>
<td>Some algorithms to decide the equivalence of recursive types, p. 26.</td>
</tr>
<tr>
<td>91/33</td>
<td>W. v.d. Aalst</td>
<td>The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.</td>
</tr>
<tr>
<td>91/34</td>
<td>J. Coenen</td>
<td>Specifying fault tolerant programs in deontic logic, p. 15.</td>
</tr>
<tr>
<td>92/01</td>
<td>J. Coenen, J. Zwiers, W.-P. de Roever</td>
<td>A note on compositional refinement, p. 27.</td>
</tr>
<tr>
<td>92/02</td>
<td>J. Coenen, J. Hooman</td>
<td>A compositional semantics for fault tolerant real-time systems, p. 18.</td>
</tr>
<tr>
<td>92/03</td>
<td>J.C.M. Baeten, J.A. Bergstra</td>
<td>Real space process algebra, p. 42.</td>
</tr>
<tr>
<td>92/05</td>
<td>J.P.H.W.v.d.Eijnde</td>
<td>Conservative fixpoint functions on a graph, p. 25.</td>
</tr>
<tr>
<td>92/06</td>
<td>J.C.M. Baeten, J.A. Bergstra</td>
<td>Discrete time process algebra, p. 45.</td>
</tr>
<tr>
<td>92/07</td>
<td>R.P. Nederpelt</td>
<td>The fine-structure of lambda calculus, p. 110.</td>
</tr>
<tr>
<td>92/10</td>
<td>P.M.P. Rambags</td>
<td>Composition and decomposition in a CPN model, p. 55.</td>
</tr>
<tr>
<td>92/13</td>
<td>F. Kamareddine</td>
<td>Set theory and nominalisation, Part II, p. 22.</td>
</tr>
<tr>
<td>92/14</td>
<td>J.C.M. Baeten</td>
<td>The total order assumption, p. 10.</td>
</tr>
<tr>
<td>92/15</td>
<td>F. Kamareddine</td>
<td>A system at the cross-roads of functional and logic programming, p. 36.</td>
</tr>
<tr>
<td>92/16</td>
<td>R.R. Seljée</td>
<td>Integrity checking in deductive databases; an exposition, p. 32.</td>
</tr>
<tr>
<td>92/17</td>
<td>W.M.P. van der Aalst</td>
<td>Interval timed coloured Petri nets and their analysis, p. 20.</td>
</tr>
<tr>
<td>92/18</td>
<td>R.Nederpelt, F. Kamareddine</td>
<td>A unified approach to Type Theory through a refined lambda-calculus, p. 30.</td>
</tr>
<tr>
<td>92/20</td>
<td>F.Kamareddine</td>
<td>Are Types for Natural Language? p. 32.</td>
</tr>
<tr>
<td>92/21</td>
<td>F.Kamareddine</td>
<td>Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.</td>
</tr>
<tr>
<td>Reference</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>92/22 R. Nederpelt F. Kamareddine</td>
<td></td>
<td>A useful lambda notation, p. 17.</td>
</tr>
<tr>
<td>92/24 M. Codish D. Dams Eyal Yardeni</td>
<td></td>
<td>Bottom-up Abstract Interpretation of Logic Programs, p. 33.</td>
</tr>
<tr>
<td>92/25 E. Poll</td>
<td></td>
<td>A Programming Logic for Fo, p. 15.</td>
</tr>
<tr>
<td>93/01 R. van Geldrop</td>
<td></td>
<td>Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.</td>
</tr>
<tr>
<td>93/02 T. Verhoeff</td>
<td></td>
<td>A continuous version of the Prisoner’s Dilemma, p. 17</td>
</tr>
<tr>
<td>93/03 T. Verhoeff</td>
<td></td>
<td>Quicksort for linked lists, p. 8.</td>
</tr>
<tr>
<td>93/04 E. H. L. Aarts J. H. M. Korst P. J. Zwietering</td>
<td></td>
<td>Deterministic and randomized local search, p. 78.</td>
</tr>
<tr>
<td>93/05 J. C. M. Baeten C. Verhoef</td>
<td></td>
<td>A congruence theorem for structured operational semantics with predicates, p. 18.</td>
</tr>
<tr>
<td>93/06 J. P. Veltkamp</td>
<td></td>
<td>On the unavoidability of metastable behaviour, p. 29.</td>
</tr>
<tr>
<td>93/07 P. D. Moerland</td>
<td></td>
<td>Exercises in Multiprogramming, p. 97.</td>
</tr>
<tr>
<td>93/08 J. Verhoosel</td>
<td></td>
<td>A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.</td>
</tr>
<tr>
<td>93/10 K. M. van Hee</td>
<td></td>
<td>Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.</td>
</tr>
<tr>
<td>Year</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>93/16</td>
<td>H. Schepers, J. Hooman</td>
<td>A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27</td>
</tr>
<tr>
<td>93/17</td>
<td>D. Altein, P. van der Stok</td>
<td>Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.</td>
</tr>
<tr>
<td>93/18</td>
<td>C. Verhoef</td>
<td>A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.</td>
</tr>
<tr>
<td>93/19</td>
<td>G-J. Houben</td>
<td>The Design of an Online Help Facility for ExSpect, p. 21.</td>
</tr>
<tr>
<td>93/22</td>
<td>E. Poll</td>
<td>A Typechecker for Bijective Pure Type Systems, p. 28.</td>
</tr>
<tr>
<td>93/23</td>
<td>E. de Kogel</td>
<td>Relational Algebra and Equational Proofs, p. 23.</td>
</tr>
<tr>
<td>93/24</td>
<td>E. Poll and Paula Severi</td>
<td>Pure Type Systems with Definitions, p. 38.</td>
</tr>
<tr>
<td>93/26</td>
<td>W.M.P. van der Aalst</td>
<td>Multi-dimensional Petri nets, p. 25.</td>
</tr>
<tr>
<td>93/27</td>
<td>T. Kloks and D. Kratsch</td>
<td>Finding all minimal separators of a graph, p. 11.</td>
</tr>
<tr>
<td>93/28</td>
<td>F. Kamareddine and R. Nederpelt</td>
<td>A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.</td>
</tr>
<tr>
<td>93/29</td>
<td>R. Post and P. De Bra</td>
<td>GOLD, a Graph Oriented Language for Databases, p. 42.</td>
</tr>
<tr>
<td>93/30</td>
<td>J. Deogun, T. Kloks, D. Kratsch, H. Müller</td>
<td>On Vertex Ranking for Permutation and Other Graphs, p. 11.</td>
</tr>
</tbody>
</table>